MATLAB®

The Language of Technical Computing

Computation

Visualization

Programming

The MATH WORKS

MATLAB Function Reference
Volume 3: P - Z

Version 6

How to Contact The MathWorks:

`www.mathworks.com`	Web
`comp.soft-sys.matlab`	Newsgroup
`support@mathworks.com`	Technical support
`suggest@mathworks.com`	Product enhancement suggestions
`bugs@mathworks.com`	Bug reports
`doc@mathworks.com`	Documentation error reports
`service@mathworks.com`	Order status, license renewals, passcodes
`info@mathworks.com`	Sales, pricing, and general information
508-647-7000	Phone
508-647-7001	Fax
The MathWorks, Inc. 3 Apple Hill Drive Natick, MA 01760-2098	Mail

For contact information about worldwide offices, see the MathWorks Web site.

MATLAB Function Reference Volume 3: P - Z

Printing History:	December 1996	First printing	(for MATLAB 5)
	June 1997	Revised for 5.1	(online version)
	October 1997	Revised for 5.2	(online version)
	January 1999	Revised for Release 11	(online version)
	June 1999	Printed for Release 11	
	March 2000	Beta	(online only)
	June 2001	Revised for 6.1	(online version)

Contents

1 Functions By Category

2 Alphabetical List of Functions

1

Functions By Category

The MATLAB Function Reference contains descriptions of all MATLAB commands and functions.

If you know the name of a function, use the “Alphabetical List of Functions” to find the reference page.

If you do not know the name of a function, select a category from the following table to see a list of related functions. You can also browse these tables to see what functionality MATLAB provides.

Category	Description
Development Environment	Startup, Command Window, help, editing and debugging, other general functions
Mathematics	Arrays and matrices, linear algebra, data analysis, other areas of mathematics
Programming and Data Types	Function/expression evaluation, program control, function handles, object oriented programming, error handling, operators, data types
File I/O	General and low-level file I/O, plus specific file formats, like audio, spreadsheet, HDF, images
Graphics	Line plots, annotating graphs, specialized plots, images, printing, Handle Graphics
3-D Visualization	Surface and mesh plots, view control, lighting and transparency, volume visualization.
Creating Graphical User Interface	GUIDE, programming graphical user interfaces.
External Interfaces	Java, ActiveX, Serial Port functions.

See Simulink, Stateflow, Real-Time Workshop, and the individual toolboxes for lists of their functions

Development Environment

General functions for working in MATLAB, including functions for startup, Command Window, help, and editing and debugging.

Category	Description
"Starting and Quitting"	Startup and shutdown options
"Command Window"	Controlling Command Window
"Getting Help"	Methods for finding information
"Workspace, File, and Search Path"	File, search path, variable management
"Programming Tools"	Editing and debugging, source control, profiling
"System"	Identifying current computer, license, or product version
"Performance Improvement Tools and Techniques"	Improving and assessing performance, e.g., memory use

Starting and Quitting

`exit`	Terminate MATLAB (same as `quit`)
`finish`	MATLAB termination M-file
`matlab`	Start MATLAB (UNIX systems only)
`matlabrc`	MATLAB startup M-file for single user systems or administrators
`quit`	Terminate MATLAB
`startup`	MATLAB startup M-file for user-defined options

Command Window

`clc`	Clear Command Window
`diary`	Save session to file
`dos`	Execute DOS command and return result
`format`	Control display format for output
`home`	Move cursor to upper left corner of Command Window
`more`	Control paged output for Command Window

`notebook`	Open M-book in Microsoft Word (Windows only)
`unix`	Execute UNIX command and return result

Getting Help

`doc`	Display online documentation in MATLAB Help browser
`docopt`	Location of help file directory for UNIX platforms
`help`	Display help for MATLAB functions in Command Window
`helpbrowser`	Display Help browser for access to extensive online help
`helpwin`	Display M-file help, with access to M-file help for all functions
`info`	Display information about The MathWorks or products
`lookfor`	Search for specified keyword in all help entries
`support`	Open MathWorks Technical Support Web page
`web`	Point Help browser or Web browser to file or Web site
`whatsnew`	Display information about MATLAB and toolbox releases

Workspace, File, and Search Path

- “Workspace”
- “File”
- “Search Path”

Workspace

`assignin`	Assign value to workspace variable
`clear`	Remove items from workspace, freeing up system memory
`evalin`	Execute string containing MATLAB expression in a workspace
`exist`	Check if variable or file exists
`openvar`	Open workspace variable in Array Editor for graphical editing
`pack`	Consolidate workspace memory
`which`	Locate functions and files
`who`, `whos`	List variables in the workspace
`workspace`	Display Workspace browser, a tool for managing the workspace

File

`cd`	Change working directory
`copyfile`	Copy file
`delete`	Delete files or graphics objects
`dir`	Display directory listing
`exist`	Check if a variable or file exists
`filebrowser`	Display Current Directory browser, a tool for viewing files
`lookfor`	Search for specified keyword in all help entries

`ls`	List directory on UNIX
`matlabroot`	Return root directory of MATLAB installation
`mkdir`	Make new directory
`pwd`	Display current directory
`rehash`	Refresh function and file system caches
`type`	List file
`what`	List MATLAB specific files in current directory
`which`	Locate functions and files

See also "File I/O" functions.

Search Path

`addpath`	Add directories to MATLAB search path
`genpath`	Generate path string
`partialpath`	Partial pathname
`path`	View or change the MATLAB directory search path
`pathtool`	Open **Set Path** dialog box to view and change MATLAB path
`rmpath`	Remove directories from MATLAB search path

Programming Tools

- "Editing and Debugging"
- "Source Control"
- "Profiling"

Editing and Debugging

`dbclear`	Clear breakpoints
`dbcont`	Resume execution
`dbdown`	Change local workspace context
`dbquit`	Quit debug mode
`dbstack`	Display function call stack
`dbstatus`	List all breakpoints
`dbstep`	Execute one or more lines from current breakpoint
`dbstop`	Set breakpoints in M-file function
`dbtype`	List M-file with line numbers
`dbup`	Change local workspace context
`edit`	Edit or create M-file
`keyboard`	Invoke the keyboard in an M-file

Source Control

`checkin`	Check file into source control system
`checkout`	Check file out of source control system
`cmopts`	Get name of source control system
`customverctrl`	Allow custom source control system
`undocheckout`	Undo previous checkout from source control system

Profiling

`profile`	Optimize performance of M-file code
`profreport`	Generate profile report

System

`computer`	Identify information about computer on which MATLAB is running
`javachk`	Generate error message based on Java feature support
`license`	Show license number for MATLAB
`usejava`	Determine if a Java feature is supported in MATLAB
`ver`	Display version information for MathWorks products
`version`	Get MATLAB version number

Performance Improvement Tools and Techniques

`memory`	Help for memory limitations
`pack`	Consolidate workspace memory
`profile`	Optimize performance of M-file code
`profreport`	Generate profile report
`rehash`	Refresh function and file system caches
`sparse`	Create sparse matrix
`zeros`	Create array of all zeros

Mathematics

Functions for working with arrays and matrices, linear algebra, data analysis, and other areas of mathematics.

Category	Description
"Arrays and Matrices"	Basic array operators and operations, creation of elementary and specialized arrays and matrices
"Linear Algebra"	Matrix analysis, linear equations, eigenvalues, singular values, logarithms, exponentials, factorization
"Elementary Math"	Trigonometry, exponentials and logarithms, complex values, rounding, remainders, discrete math
"Data Analysis and Fourier Transforms"	Descriptive statistics, finite differences, correlation, filtering and convolution, fourier transforms
"Polynomials"	Multiplication, division, evaluation, roots, derivatives, integration, eigenvalue problem, curve fitting, partial fraction expansion
"Interpolation and Computational Geometry"	Interpolation, Delaunay triangulation and tessellation, convex hulls, Voronoi diagrams, domain generation
"Coordinate System Conversion"	Conversions between Cartesian and polar or spherical coordinates
"Nonlinear Numerical Methods"	Differential equations, optimization, integration
"Specialized Math"	Airy, Bessel, Jacobi, Legendre, beta, elliptic, error, exponential integral, gamma functions

Category	Description
"Sparse Matrices"	Elementary sparse matrices, operations, reordering algorithms, linear algebra, iterative methods, tree operations
"Math Constants"	Pi, imaginary unit, infinity, Not-a-Number, largest and smallest positive floating point numbers, floating point relative accuracy

Arrays and Matrices

- "Basic Information"
- "Operators"
- "Operations and Manipulation"
- "Elementary Matrices and Arrays"
- "Specialized Matrices"

Basic Information

`disp`	Display array
`display`	Display array
`isempty`	True for empty matrix
`isequal`	True if arrays are identical
`islogical`	True for logical array
`isnumeric`	True for numeric arrays
`issparse`	True for sparse matrix
`length`	Length of vector
`ndims`	Number of dimensions
`numel`	Number of elements
`size`	Size of matrix

Operators

`+`	Addition
`+`	Unary plus
`-`	Subtraction
`-`	Unary minus
`*`	Matrix multiplication
`^`	Matrix power
`\`	Backslash or left matrix divide

`/`	Slash or right matrix divide
`'`	Transpose
`.'`	Nonconjugated transpose
`.*`	Array multiplication (element-wise)
`.^`	Array power (element-wise)
`.\`	Left array divide (element-wise)
`./`	Right array divide (element-wise)

Operations and Manipulation

`:` (colon)	Index into array, rearrange array
`blkdiag`	Block diagonal concatenation
`cat`	Concatenate arrays
`cross`	Vector cross product
`cumprod`	Cumulative product
`cumsum`	Cumulative sum
`diag`	Diagonal matrices and diagonals of matrix
`dot`	Vector dot product
`end`	Last index
`find`	Find indices of nonzero elements
`fliplr`	Flip matrices left-right
`flipud`	Flip matrices up-down
`flipdim`	Flip matrix along specified dimension
`horzcat`	Horizontal concatenation
`ind2sub`	Multiple subscripts from linear index
`ipermute`	Inverse permute dimensions of multidimensional array
`kron`	Kronecker tensor product
`max`	Maximum elements of array
`min`	Minimum elements of array
`permute`	Rearrange dimensions of multidimensional array
`prod`	Product of array elements
`repmat`	Replicate and tile array
`reshape`	Reshape array
`rot90`	Rotate matrix 90 degrees
`sort`	Sort elements in ascending order
`sortrows`	Sort rows in ascending order
`sum`	Sum of array elements
`sqrtm`	Matrix square root
`sub2ind`	Linear index from multiple subscripts
`tril`	Lower triangular part of matrix
`triu`	Upper triangular part of matrix
`vertcat`	Vertical concatenation

See also "Linear Algebra" for other matrix operations.
See also "Elementary Math" for other array operations.

Elementary Matrices and Arrays

`:` (colon)	Regularly spaced vector
`blkdiag`	Construct block diagonal matrix from input arguments
`diag`	Diagonal matrices and diagonals of matrix
`eye`	Identity matrix
`freqspace`	Frequency spacing for frequency response
`linspace`	Generate linearly spaced vectors
`logspace`	Generate logarithmically spaced vectors
`meshgrid`	Generate X and Y matrices for three-dimensional plots
`ndgrid`	Arrays for multidimensional functions and interpolation
`ones`	Create array of all ones
`rand`	Uniformly distributed random numbers and arrays
`randn`	Normally distributed random numbers and arrays
`repmat`	Replicate and tile array
`zeros`	Create array of all zeros

Specialized Matrices

`compan`	Companion matrix
`gallery`	Test matrices
`hadamard`	Hadamard matrix
`hankel`	Hankel matrix
`hilb`	Hilbert matrix
`invhilb`	Inverse of Hilbert matrix
`magic`	Magic square
`pascal`	Pascal matrix
`rosser`	Classic symmetric eigenvalue test problem
`toeplitz`	Toeplitz matrix
`vander`	Vandermonde matrix
`wilkinson`	Wilkinson's eigenvalue test matrix

Linear Algebra

- "Matrix Analysis"
- "Linear Equations"
- "Eigenvalues and Singular Values"
- "Matrix Logarithms and Exponentials"
- "Factorization"

Matrix Analysis

`cond`	Condition number with respect to inversion
`condeig`	Condition number with respect to eigenvalues
`det`	Determinant
`norm`	Matrix or vector norm
`normest`	Estimate matrix 2-norm
`null`	Null space
`orth`	Orthogonalization
`rank`	Matrix rank
`rcond`	Matrix reciprocal condition number estimate
`rref`	Reduced row echelon form
`subspace`	Angle between two subspaces
`trace`	Sum of diagonal elements

Linear Equations

`\` and `/`	Linear equation solution
`chol`	Cholesky factorization
`cholinc`	Incomplete Cholesky factorization
`cond`	Condition number with respect to inversion
`condest`	1-norm condition number estimate
`funm`	Evaluate general matrix function
`inv`	Matrix inverse
`lscov`	Least squares solution in presence of known covariance
`lsqnonneg`	Nonnegative least squares
`lu`	LU matrix factorization
`luinc`	Incomplete LU factorization
`pinv`	Moore-Penrose pseudoinverse of matrix
`qr`	Orthogonal-triangular decomposition
`rcond`	Matrix reciprocal condition number estimate

Eigenvalues and Singular Values

`balance`	Improve accuracy of computed eigenvalues
`cdf2rdf`	Convert complex diagonal form to real block diagonal form
`condeig`	Condition number with respect to eigenvalues
`eig`	Eigenvalues and eigenvectors
`eigs`	Eigenvalues and eigenvectors of sparse matrix
`gsvd`	Generalized singular value decomposition
`hess`	Hessenberg form of matrix
`poly`	Polynomial with specified roots
`polyeig`	Polynomial eigenvalue problem
`qz`	QZ factorization for generalized eigenvalues
`rsf2csf`	Convert real Schur form to complex Schur form

`schur`	Schur decomposition
`svd`	Singular value decomposition
`svds`	Singular values and vectors of sparse matrix

Matrix Logarithms and Exponentials

`expm`	Matrix exponential
`logm`	Matrix logarithm
`sqrtm`	Matrix square root

Factorization

`balance`	Diagonal scaling to improve eigenvalue accuracy
`cdf2rdf`	Complex diagonal form to real block diagonal form
`chol`	Cholesky factorization
`cholinc`	Incomplete Cholesky factorization
`cholupdate`	Rank 1 update to Cholesky factorization
`lu`	LU matrix factorization
`luinc`	Incomplete LU factorization
`planerot`	Givens plane rotation
`qr`	Orthogonal-triangular decomposition
`qrdelete`	Delete column from QR factorization
`qrinsert`	Insert column in QR factorization
`qrupdate`	Rank 1 update to QR factorization
`qz`	QZ factorization for generalized eigenvalues
`rsf2csf`	Real block diagonal form to complex diagonal form

Elementary Math

- “Trigonometric”
- “Exponential”
- “Complex”
- “Rounding and Remainder”
- “Discrete Math (e.g., Prime Factors)”

Trigonometric

`acos`, `acosh`	Inverse cosine and inverse hyperbolic cosine
`acot`, `acoth`	Inverse cotangent and inverse hyperbolic cotangent
`acsc`, `acsch`	Inverse cosecant and inverse hyperbolic cosecant
`asec`, `asech`	Inverse secant and inverse hyperbolic secant
`asin`, `asinh`	Inverse sine and inverse hyperbolic sine

`atan`, `atanh`	Inverse tangent and inverse hyperbolic tangent
`atan2`	Four-quadrant inverse tangent
`cos`, `cosh`	Cosine and hyperbolic cosine
`cot`, `coth`	Cotangent and hyperbolic cotangent
`csc`, `csch`	Cosecant and hyperbolic cosecant
`sec`, `sech`	Secant and hyperbolic secant
`sin`, `sinh`	Sine and hyperbolic sine
`tan`, `tanh`	Tangent and hyperbolic tangent

Exponential

`exp`	Exponential
`log`	Natural logarithm
`log2`	Base 2 logarithm and dissect floating-point numbers into exponent and mantissa
`log10`	Common (base 10) logarithm
`nextpow2`	Next higher power of 2
`pow2`	Base 2 power and scale floating-point number
`sqrt`	Square root

Complex

`abs`	Absolute value
`angle`	Phase angle
`complex`	Construct complex data from real and imaginary parts
`conj`	Complex conjugate
`cplxpair`	Sort numbers into complex conjugate pairs
i	Imaginary unit
`imag`	Complex imaginary part
`isreal`	True for real array
`j`	Imaginary unit
`real`	Complex real part
`unwrap`	Unwrap phase angle

Rounding and Remainder

`fix`	Round towards zero
`floor`	Round towards minus infinity
`ceil`	Round towards plus infinity
`round`	Round towards nearest integer
`mod`	Modulus (signed remainder after division)
`rem`	Remainder after division
`sign`	Signum

Discrete Math (e.g., Prime Factors)

`factor`	Prime factors
`factorial`	Factorial function
`gcd`	Greatest common divisor
`isprime`	True for prime numbers
`lcm`	Least common multiple
`nchoosek`	All combinations of N elements taken K at a time
`perms`	All possible permutations
`primes`	Generate list of prime numbers
`rat`, `rats`	Rational fraction approximation

Data Analysis and Fourier Transforms

- "Basic Operations"
- "Finite Differences"
- "Correlation"
- "Filtering and Convolution"
- "Fourier Transforms"

Basic Operations

`cumprod`	Cumulative product
`cumsum`	Cumulative sum
`cumtrapz`	Cumulative trapezoidal numerical integration
`max`	Maximum elements of array
`mean`	Average or mean value of arrays
`median`	Median value of arrays
`min`	Minimum elements of array
`prod`	Product of array elements
`sort`	Sort elements in ascending order
`sortrows`	Sort rows in ascending order
`std`	Standard deviation
`sum`	Sum of array elements
`trapz`	Trapezoidal numerical integration
`var`	Variance

Finite Differences

`del2`	Discrete Laplacian
`diff`	Differences and approximate derivatives
`gradient`	Numerical gradient

Correlation

`corrcoef`	Correlation coefficients
`cov`	Covariance matrix
`subspace`	Angle between two subspaces

Filtering and Convolution

`conv`	Convolution and polynomial multiplication
`conv2`	Two-dimensional convolution
`convn`	N-dimensional convolution
`deconv`	Deconvolution and polynomial division
`detrend`	Linear trend removal
`filter`	Filter data with infinite impulse response (IIR) or finite impulse response (FIR) filter
`filter2`	Two-dimensional digital filtering

Fourier Transforms

`abs`	Absolute value and complex magnitude
`angle`	Phase angle
`fft`	One-dimensional fast Fourier transform
`fft2`	Two-dimensional fast Fourier transform
`fftn`	N-dimensional discrete Fourier Transform
`fftshift`	Shift DC component of fast Fourier transform to center of spectrum
`ifft`	Inverse one-dimensional fast Fourier transform
`ifft2`	Inverse two-dimensional fast Fourier transform
`ifftn`	Inverse multidimensional fast Fourier transform
`ifftshift`	Inverse fast Fourier transform shift
`nextpow2`	Next power of two
`unwrap`	Correct phase angles

Polynomials

`conv`	Convolution and polynomial multiplication
`deconv`	Deconvolution and polynomial division
`poly`	Polynomial with specified roots
`polyder`	Polynomial derivative
`polyeig`	Polynomial eigenvalue problem
`polyfit`	Polynomial curve fitting
`polyint`	Analytic polynomial integration
`polyval`	Polynomial evaluation
`polyvalm`	Matrix polynomial evaluation
`residue`	Convert between partial fraction expansion and polynomial coefficients
`roots`	Polynomial roots

Interpolation and Computational Geometry

- "Interpolation"
- "Delaunay Triangulation and Tessellation"
- "Convex Hull"
- "Voronoi Diagrams"
- "Domain Generation"

Interpolation

`dsearch`	Search for nearest point
`dsearchn`	Multidimensional closest point search
`griddata`	Data gridding
`griddata3`	Data gridding and hypersurface fitting for three-dimensional data
`griddatan`	Data gridding and hypersurface fitting (dimension >= 2)
`interp1`	One-dimensional data interpolation (table lookup)
`interp2`	Two-dimensional data interpolation (table lookup)
`interp3`	Three-dimensional data interpolation (table lookup)
`interpft`	One-dimensional interpolation using fast Fourier transform method
`interpn`	Multidimensional data interpolation (table lookup)
`meshgrid`	Generate X and Y matrices for three-dimensional plots
`mkpp`	Make piecewise polynomial
`ndgrid`	Generate arrays for multidimensional functions and interpolation
`pchip`	Piecewise Cubic Hermite Interpolating Polynomial (PCHIP)
`ppval`	Piecewise polynomial evaluation
`spline`	Cubic spline data interpolation
`tsearchn`	Multidimensional closest simplex search
`unmkpp`	Piecewise polynomial details

Delaunay Triangulation and Tessellation

`delaunay`	Delaunay triangulation
`delaunay3`	Three-dimensional Delaunay tessellation
`delaunayn`	Multidimensional Delaunay tessellation
`dsearch`	Search for nearest point
`dsearchn`	Multidimensional closest point search
`tetramesh`	Tetrahedron mesh plot
`trimesh`	Triangular mesh plot
`triplot`	Two-dimensional triangular plot
`trisurf`	Triangular surface plot
`tsearch`	Search for enclosing Delaunay triangle
`tsearchn`	Multidimensional closest simplex search

Convex Hull

`convhull`	Convex hull
`convhulln`	Multidimensional convex hull
`patch`	Create patch graphics object
`plot`	Linear two-dimensional plot
`trisurf`	Triangular surface plot

Voronoi Diagrams

`dsearch`	Search for nearest point
`patch`	Create patch graphics object
`plot`	Linear two-dimensional plot
`voronoi`	Voronoi diagram
`voronoin`	Multidimensional Voronoi diagrams

Domain Generation

`meshgrid`	Generate X and Y matrices for three-dimensional plots
`ndgrid`	Generate arrays for multidimensional functions and interpolation

Coordinate System Conversion

Cartesian

`cart2sph`	Transform Cartesian to spherical coordinates
`cart2pol`	Transform Cartesian to polar coordinates
`pol2cart`	Transform polar to Cartesian coordinates
`sph2cart`	Transform spherical to Cartesian coordinates

Nonlinear Numerical Methods

- "Ordinary Differential Equations (IVP)"
- "Boundary Value Problems"
- "Partial Differential Equations"
- "Optimization"
- "Numerical Integration (Quadrature)"

Ordinary Differential Equations (IVP)

`deval`	Evaluate solution of differential equation problem
`ode113`	Solve non-stiff differential equations, variable order method
`ode15s`	Solve stiff ODEs and DAEs Index 1, variable order method

`ode23`	Solve non-stiff differential equations, low order method
`ode23s`	Solve stiff differential equations, low order method
`ode23t`	Solve moderately stiff ODEs and DAEs Index 1, trapezoidal rule
`ode23tb`	Solve stiff differential equations, low order method
`ode45`	Solve non-stiff differential equations, medium order method
`odeget`	Get ODE `options` parameters
`odeset`	Create/alter ODE `options` structure

Boundary Value Problems

`bvp4c`	Solve two-point boundary value problems for ODEs by collocation
`bvpset`	Create/alter BVP `options` structure
`bvpget`	Get BVP `options` parameters
`deval`	Evaluate solution of differential equation problem

Partial Differential Equations

`pdepe`	Solve initial-boundary value problems for parabolic-elliptic PDEs
`pdeval`	Evaluates by interpolation solution computed by `pdepe`

Optimization

`fminbnd`	Scalar bounded nonlinear function minimization
`fminsearch`	Multidimensional unconstrained nonlinear minimization, by Nelder-Mead direct search method
`fzero`	Scalar nonlinear zero finding
`lsqnonneg`	Linear least squares with nonnegativity constraints
`optimset`	Create or alter optimization `options` structure
`optimget`	Get optimization parameters from `options` structure

Numerical Integration (Quadrature)

`quad`	Numerically evaluate integral, adaptive Simpson quadrature (low order)
`quadl`	Numerically evaluate integral, adaptive Lobatto quadrature (high order)
`dblquad`	Numerically evaluate double integral

Specialized Math

`airy`	Airy functions
`besselh`	Bessel functions of third kind (Hankel functions)
`besseli`	Modified Bessel function of first kind
`besselj`	Bessel function of first kind
`besselk`	Modified Bessel function of second kind
`bessely`	Bessel function of second kind
`beta`	Beta function

`betainc`	Incomplete beta function
`betaln`	Logarithm of beta function
`ellipj`	Jacobi elliptic functions
`ellipke`	Complete elliptic integrals of first and second kind
`erf`	Error function
`erfc`	Complementary error function
`erfcinv`	Inverse complementary error function
`erfcx`	Scaled complementary error function
`erfinv`	Inverse error function
`expint`	Exponential integral
`gamma`	Gamma function
`gammainc`	Incomplete gamma function
`gammaln`	Logarithm of gamma function
`legendre`	Associated Legendre functions

Sparse Matrices

- "Elementary Sparse Matrices"
- "Full to Sparse Conversion"
- "Working with Sparse Matrices"
- "Reordering Algorithms"
- "Linear Algebra"
- "Linear Equations (Iterative Methods)"
- "Tree Operations"

Elementary Sparse Matrices

`spdiags`	Sparse matrix formed from diagonals
`speye`	Sparse identity matrix
`sprand`	Sparse uniformly distributed random matrix
`sprandn`	Sparse normally distributed random matrix
`sprandsym`	Sparse random symmetric matrix

Full to Sparse Conversion

`find`	Find indices of nonzero elements
`full`	Convert sparse matrix to full matrix
`sparse`	Create sparse matrix
`spconvert`	Import from sparse matrix external format

Working with Sparse Matrices

`issparse`	True for sparse matrix
`nnz`	Number of nonzero matrix elements
`nonzeros`	Nonzero matrix elements
`nzmax`	Amount of storage allocated for nonzero matrix elements
`spalloc`	Allocate space for sparse matrix
`spfun`	Apply function to nonzero matrix elements
`spones`	Replace nonzero sparse matrix elements with ones
`spparms`	Set parameters for sparse matrix routines
`spy`	Visualize sparsity pattern

Reordering Algorithms

`colamd`	Column approximate minimum degree permutation
`colmmd`	Column minimum degree permutation
`colperm`	Column permutation
`dmperm`	Dulmage-Mendelsohn permutation
`randperm`	Random permutation
`symamd`	Symmetric approximate minimum degree permutation
`symmmd`	Symmetric minimum degree permutation
`symrcm`	Symmetric reverse Cuthill-McKee permutation

Linear Algebra

`cholinc`	Incomplete Cholesky factorization
`condest`	1-norm condition number estimate
`eigs`	Eigenvalues and eigenvectors of sparse matrix
`luinc`	Incomplete LU factorization
`normest`	Estimate matrix 2-norm
`sprank`	Structural rank
`svds`	Singular values and vectors of sparse matrix

Linear Equations (Iterative Methods)

`bicg`	BiConjugate Gradients method
`bicgstab`	BiConjugate Gradients Stabilized method
`cgs`	Conjugate Gradients Squared method
`gmres`	Generalized Minimum Residual method
`lsqr`	LSQR implementation of Conjugate Gradients on Normal Equations
`minres`	Minimum Residual method
`pcg`	Preconditioned Conjugate Gradients method
`qmr`	Quasi-Minimal Residual method
`spaugment`	Form least squares augmented system
`symmlq`	Symmetric LQ method

Tree Operations

`etree`	Elimination tree
`etreeplot`	Plot elimination tree
`gplot`	Plot graph, as in "graph theory"
`symbfact`	Symbolic factorization analysis
`treelayout`	Lay out tree or forest
`treeplot`	Plot picture of tree

Math Constants

`eps`	Floating-point relative accuracy
`i`	Imaginary unit
`Inf`	Infinity, ∞
`j`	Imaginary unit
`NaN`	Not-a-Number
`pi`	Ratio of a circle's circumference to its diameter, π
`realmax`	Largest positive floating-point number
`realmin`	Smallest positive floating-point number

Programming and Data Types

Functions to store and operate on data at either the MATLAB command line or in programs and scripts. Functions to write, manage, and execute MATLAB programs.

Category	Description
"Data Types"	Numeric, character, structures, cell arrays, and data type conversion
"Arrays"	Basic array operations and manipulation
"Operators and Operations"	Special characters and arithmetic, bit-wise, relational, logical, set, date and time operations
"Programming in MATLAB"	M-files, function/expression evaluation, program control, function handles, object oriented programming, error handling

Data Types

- "Numeric"
- "Characters and Strings"
- "Structures"
- "Cell Arrays"
- "Data Type Conversion"

Numeric

`[ ]`	Array constructor
`cat`	Concatenate arrays
`class`	Return object's class name (e.g., numeric)
`find`	Find indices and values of nonzero array elements
`ipermute`	Inverse permute dimensions of multidimensional array
`isa`	Detect object of given class (e.g., numeric)
`isequal`	Determine if arrays are numerically equal
`isnumeric`	Determine if item is numeric array
`isreal`	Determine if all array elements are real numbers

`permute`	Rearrange dimensions of multidimensional array
`reshape`	Reshape array
`squeeze`	Remove singleton dimensions from array
`zeros`	Create array of all zeros

Characters and Strings

Description of Strings in MATLAB

`strings`	Describes MATLAB string handling

Creating and Manipulating Strings

`blanks`	Create string of blanks
`char`	Create character array (string)
`cellstr`	Create cell array of strings from character array
`datestr`	Convert to date string format
`deblank`	Strip trailing blanks from the end of string
`lower`	Convert string to lower case
`sprintf`	Write formatted data to string
`sscanf`	Read string under format control
`strcat`	String concatenation
`strjust`	Justify character array
`strread`	Read formatted data from string
`strrep`	String search and replace
`strvcat`	Vertical concatenation of strings
`upper`	Convert string to upper case

Comparing and Searching Strings

`class`	Return object's class name (e.g., char)
`findstr`	Find string within another, longer string
`isa`	Detect object of given class (e.g., char)
`iscellstr`	Determine if item is cell array of strings
`ischar`	Determine if item is character array
`isletter`	Detect array elements that are letters of the alphabet
`isspace`	Detect elements that are ASCII white spaces
`strcmp`	Compare strings
`strcmpi`	Compare strings, ignoring case
`strfind`	Find one string within another
`strmatch`	Find possible matches for string
`strncmp`	Compare first `n` characters of strings
`strncmpi`	Compare first `n` characters of strings, ignoring case
`strtok`	First token in string

Evaluating String Expressions

`eval`	Execute string containing MATLAB expression
`evalc`	Evaluate MATLAB expression with capture
`evalin`	Execute string containing MATLAB expression in workspace

Structures

`cell2struct`	Cell array to structure array conversion
`class`	Return object's class name (e.g., struct)
`deal`	Deal inputs to outputs
`fieldnames`	Field names of structure
`getfield`	Get field of structure array
`isa`	Detect object of given class (e.g., struct)
`isequal`	Determine if arrays are numerically equal
`isfield`	Determine if item is structure array field
`isstruct`	Determine if item is structure array
`rmfield`	Remove structure fields
`setfield`	Set field of structure array
`struct`	Create structure array
`struct2cell`	Structure to cell array conversion

Cell Arrays

`{ }`	Construct cell array
`cell`	Construct cell array
`cellfun`	Apply function to each element in cell array
`cellstr`	Create cell array of strings from character array
`cell2struct`	Cell array to structure array conversion
`celldisp`	Display cell array contents
`cellplot`	Graphically display structure of cell arrays
`class`	Return object's class name (e.g., cell)
`deal`	Deal inputs to outputs
`isa`	Detect object of given class (e.g., cell)
`iscell`	Determine if item is cell array
`iscellstr`	Determine if item is cell array of strings
`isequal`	Determine if arrays are numerically equal
`num2cell`	Convert numeric array into cell array
`struct2cell`	Structure to cell array conversion

Data Type Conversion

Numeric

`double`	Convert to double-precision

`int8`	Convert to signed 8-bit integer
`int16`	Convert to signed 16-bit integer
`int32`	Convert to signed 32-bit integer
`single`	Convert to single-precision
`uint8`	Convert to unsigned 8-bit integer
`uint16`	Convert to unsigned 16-bit integer
`uint32`	Convert to unsigned 32-bit integer

String to Numeric

`base2dec`	Convert base `N` number string to decimal number
`bin2dec`	Convert binary number string to decimal number
`hex2dec`	Convert hexadecimal number string to decimal number
`hex2num`	Convert hexadecimal number string to double number
`str2double`	Convert string to double-precision number
`str2num`	Convert string to number

Numeric to String

`char`	Convert to character array (string)
`dec2base`	Convert decimal to base `N` number in string
`dec2bin`	Convert decimal to binary number in string
`dec2hex`	Convert decimal to hexadecimal number in string
`int2str`	Convert integer to string
`mat2str`	Convert a matrix to string
`num2str`	Convert number to string

Other Conversions

`cell2struct`	Convert cell array to structure array
`datestr`	Convert serial date number to string
`func2str`	Convert function handle to function name string
`logical`	Convert numeric to logical array
`num2cell`	Convert a numeric array to cell array
`str2func`	Convert function name string to function handle
`struct2cell`	Convert structure to cell array

Determine Data Type

`is*`	Detect state
`isa`	Detect object of given MATLAB class or Java class
`iscell`	Determine if item is cell array
`iscellstr`	Determine if item is cell array of strings
`ischar`	Determine if item is character array
`isfield`	Determine if item is character array

`isjava`	Determine if item is Java object
`islogical`	Determine if item is logical array
`isnumeric`	Determine if item is numeric array
`isobject`	Determine if item is MATLAB OOPs object
`isstruct`	Determine if item is MATLAB structure array

Arrays

- "Array Operations"
- "Basic Array Information"
- "Array Manipulation"
- "Elementary Arrays"

Array Operations

`[ ]`	Array constructor
`,`	Array row element separator
`;`	Array column element separator
`:`	Specify range of array elements
`end`	Indicate last index of array
`+`	Addition or unary plus
`-`	Subtraction or unary minus
`.*`	Array multiplication
`./`	Array right division
`.\`	Array left division
`.^`	Array power
`.'`	Array (nonconjugated) transpose

Basic Array Information

`disp`	Display text or array
`display`	Overloaded method to display text or array
`isempty`	Determine if array is empty
`isequal`	Determine if arrays are numerically equal
`isnumeric`	Determine if item is numeric array
`islogical`	Determine if item is logical array
`length`	Length of vector
`ndims`	Number of array dimensions
`numel`	Number of elements in matrix or cell array
`size`	Array dimensions

Array Manipulation

`:`	Specify range of array elements
`blkdiag`	Construct block diagonal matrix from input arguments
`cat`	Concatenate arrays
`find`	Find indices and values of nonzero elements
`fliplr`	Flip matrices left-right
`flipud`	Flip matrices up-down
`flipdim`	Flip array along specified dimension
`horzcat`	Horizontal concatenation
`ind2sub`	Subscripts from linear index
`ipermute`	Inverse permute dimensions of multidimensional array
`permute`	Rearrange dimensions of multidimensional array
`repmat`	Replicate and tile array
`reshape`	Reshape array
`rot90`	Rotate matrix 90 degrees
`shiftdim`	Shift dimensions
`sort`	Sort elements in ascending order
`sortrows`	Sort rows in ascending order
`squeeze`	Remove singleton dimensions
`sub2ind`	Single index from subscripts
`vertcat`	Horizontal concatenation

Elementary Arrays

`:`	Regularly spaced vector
`blkdiag`	Construct block diagonal matrix from input arguments
`eye`	Identity matrix
`linspace`	Generate linearly spaced vectors
`logspace`	Generate logarithmically spaced vectors
`meshgrid`	Generate X and Y matrices for three-dimensional plots
`ndgrid`	Generate arrays for multidimensional functions and interpolation
`ones`	Create array of all ones
`rand`	Uniformly distributed random numbers and arrays
`randn`	Normally distributed random numbers and arrays
`zeros`	Create array of all zeros

Operators and Operations

- "Special Characters"
- "Arithmetic Operations"
- "Bit-wise Operations"
- "Relational Operations"

- "Logical Operations"
- "Set Operations"
- "Date and Time Operations"

Special Characters

:	Specify range of array elements
( )	Pass function arguments, or prioritize operations
[ ]	Construct array
{ }	Construct cell array
.	Decimal point, or structure field separator
...	Continue statement to next line
,	Array row element separator
;	Array column element separator
%	Insert comment line into code
!	Command to operating system
=	Assignment

Arithmetic Operations

+	Plus
-	Minus
.	Decimal point
=	Assignment
*	Matrix multiplication
/	Matrix right division
\	Matrix left division
^	Matrix power
'	Matrix transpose
.*	Array multiplication (element-wise)
./	Array right division (element-wise)
.\	Array left division (element-wise)
.^	Array power (element-wise)
.'	Array transpose

Bit-wise Operations

bitand	Bit-wise AND
bitcmp	Bit-wise complement
bitor	Bit-wise OR
bitmax	Maximum floating-point integer
bitset	Set bit at specified position
bitshift	Bit-wise shift
bitget	Get bit at specified position

`bitxor`	Bit-wise XOR

Relational Operations

`<`	Less than
`<=`	Less than or equal to
`>`	Greater than
`>=`	Greater than or equal to
`==`	Equal to
`~=`	Not equal to

Logical Operations

`&`	Logical AND
`|`	Logical OR
`~`	Logical NOT
`all`	Test to determine if all elements are nonzero
`any`	Test for any nonzero elements
`find`	Find indices and values of nonzero elements
`is*`	Detect state
`isa`	Detect object of given class
`iskeyword`	Determine if string is MATLAB keyword
`isvarname`	Determine if string is valid variable name
`logical`	Convert numeric values to logical
`xor`	Logical EXCLUSIVE OR

Set Operations

`intersect`	Set intersection of two vectors
`ismember`	Detect members of set
`setdiff`	Return set difference of two vectors
`setxor`	Set exclusive or of two vectors
`union`	Set union of two vectors
`unique`	Unique elements of vector

Date and Time Operations

`calendar`	Calendar for specified month
`clock`	Current time as date vector
`cputime`	Elapsed CPU time
`date`	Current date string
`datenum`	Serial date number
`datestr`	Convert serial date number to string
`datevec`	Date components
`eomday`	End of month

`etime`	Elapsed time
`now`	Current date and time
`tic, toc`	Stopwatch timer
`weekday`	Day of the week

Programming in MATLAB

- "M-File Functions and Scripts"
- "Evaluation of Expressions and Functions"
- "Variables and Functions in Memory"
- "Control Flow"
- "Function Handles"
- "Object-Oriented Programming"
- "Error Handling"
- "MEX Programming"

M-File Functions and Scripts

`( )`	Pass function arguments
`%`	Insert comment line into code
`...`	Continue statement to next line
`depfun`	List dependent functions of M-file or P-file
`depdir`	List dependent directories of M-file or P-file
`function`	Function M-files
`input`	Request user input
`inputname`	Input argument name
`mfilename`	Name of currently running M-file
`nargin`	Number of function input arguments
`nargout`	Number of function output arguments
`nargchk`	Check number of input arguments
`nargoutchk`	Validate number of output arguments
`pcode`	Create preparsed pseudocode file (P-file)
`script`	Describes script M-file
`varargin`	Accept variable number of arguments
`varargout`	Return variable number of arguments

Evaluation of Expressions and Functions

`builtin`	Execute builtin function from overloaded method
`cellfun`	Apply function to each element in cell array
`eval`	Interpret strings containing MATLAB expressions

`evalc`	Evaluate MATLAB expression with capture
`evalin`	Evaluate expression in workspace
`feval`	Evaluate function
`iskeyword`	Determine if item is MATLAB keyword
`isvarname`	Determine if item is valid variable name
`pause`	Halt execution temporarily
`run`	Run script that is not on current path
`script`	Describes script M-file
`symvar`	Determine symbolic variables in expression
`tic`, `toc`	Stopwatch timer

Variables and Functions in Memory

`assignin`	Assign value to workspace variable
`global`	Define global variables
`inmem`	Return names of functions in memory
`isglobal`	Determine if item is global variable
`mislocked`	True if M-file cannot be cleared
`mlock`	Prevent clearing M-file from memory
`munlock`	Allow clearing M-file from memory
`pack`	Consolidate workspace memory
`persistent`	Define persistent variable
`rehash`	Refresh function and file system caches

Control Flow

`break`	Terminate execution of `for` loop or `while` loop
`case`	Case switch
`catch`	Begin catch block
`continue`	Pass control to next iteration of `for` or `while` loop
`else`	Conditionally execute statements
`elseif`	Conditionally execute statements
`end`	Terminate conditional statements, or indicate last index
`error`	Display error messages
`for`	Repeat statements specific number of times
`if`	Conditionally execute statements
`otherwise`	Default part of `switch` statement
`return`	Return to invoking function
`switch`	Switch among several cases based on expression
`try`	Begin `try` block
`while`	Repeat statements indefinite number of times

Function Handles

`class`	Return object’s class name (e.g. function_handle)

`feval`	Evaluate function
`function_handle`	Describes function handle data type
`functions`	Return information about function handle
`func2str`	Constructs function name string from function handle
`isa`	Detect object of given class (e.g. function_handle)
`isequal`	Determine if function handles are equal
`str2func`	Constructs function handle from function name string

Object-Oriented Programming

MATLAB Classes and Objects

`class`	Create object or return class of object
`fieldnames`	List public fields belonging to object,
`inferiorto`	Establish inferior class relationship
`isa`	Detect object of given class
`isobject`	Determine if item is MATLAB OOPs object
`loadobj`	User-defined extension of `load` function for user objects
`methods`	Display method names
`methodsview`	Displays information on all methods implemented by class
`saveobj`	User-defined extension of `save` function for user objects
`subsasgn`	Overloaded method for `A(I)=B`, `A{I}=B`, and `A.field=B`
`subsindex`	Overloaded method for `X(A)`
`subsref`	Overloaded method for `A(I)`, `A{I}` and `A.field`
`substruct`	Create structure argument for subsasgn or subsref
`superiorto`	Establish superior class relationship

Java Classes and Objects

`cell`	Convert Java array object to cell array
`class`	Return class name of Java object
`clear`	Clear Java packages import list
`depfun`	List Java classes used by M-file
`exist`	Detect if item is Java class
`fieldnames`	List public fields belonging to object,
`import`	Add package or class to current Java import list
`inmem`	List names of Java classes loaded into memory
`isa`	Detect object of given class
`isjava`	Determine whether object is Java object
`javaArray`	Constructs Java array
`javaMethod`	Invokes Java method
`javaObject`	Constructs Java object
`methods`	Display methods belonging to class

`methodsview`	Display information on all methods implemented by class
`which`	Display package and class name for method

Error Handling

`catch`	Begin `catch` block of `try/catch` statement
`error`	Display error message
`ferror`	Query MATLAB about errors in file input or output
`lasterr`	Return last error message generated by MATLAB
`lastwarn`	Return last warning message issued by MATLAB
`try`	Begin `try` block of `try/catch` statement
`warning`	Display warning message

MEX Programming

`dbmex`	Enable MEX-file debugging
`inmem`	Return names of currently loaded MEX-files
`mex`	Compile MEX-function from C or Fortran source code
`mexext`	Return MEX-filename extension

File I/O

Functions to read and write data to files of different format types.

Category	Description
"Filename Construction"	Get path, directory, filename information; construct filenames
"Opening, Loading, Saving Files"	Open files; transfer data between files and MATLAB workspace
"Low-Level File I/O"	Low-level operations that use a file identifier (e.g., fopen, fseek, fread)
"Text Files"	Delimited or formatted I/O to text files
"Spreadsheets"	Excel and Lotus 123 files
"Scientific Data"	CDF, FITS, HDF formats
"Audio and Audio/Video"	General audio functions; SparcStation, Wave, AVI files
"Images"	Graphics files

To see a listing of file formats that are readable from MATLAB, go to `file formats`.

Filename Construction

`fileparts`	Return parts of filename
`filesep`	Return directory separator for this platform
`fullfile`	Build full filename from parts
`tempdir`	Return name of system's temporary directory
`tempname`	Return unique string for use as temporary filename

Opening, Loading, Saving Files

`importdata`	Load data from various types of files
`load`	Load all or specific data from MAT or ASCII file

`open`	Open files of various types using appropriate editor or program
`save`	Save all or specific data to MAT or ASCII file

Low-Level File I/O

`fclose`	Close one or more open files
`feof`	Test for end-of-file
`ferror`	Query MATLAB about errors in file input or output
`fgetl`	Return next line of file as string without line terminator(s)
`fgets`	Return next line of file as string with line terminator(s)
`fopen`	Open file or obtain information about open files
`fprintf`	Write formatted data to file
`fread`	Read binary data from file
`frewind`	Rewind open file
`fscanf`	Read formatted data from file
`fseek`	Set file position indicator
`ftell`	Get file position indicator
`fwrite`	Write binary data to file

Text Files

`csvread`	Read numeric data from text file, using comma delimiter
`csvwrite`	Write numeric data to text file, using comma delimiter
`dlmread`	Read numeric data from text file, specifying your own delimiter
`dlmwrite`	Write numeric data to text file, specifying your own delimiter
`textread`	Read data from text file, specifying format for each value

Spreadsheets

Microsoft Excel Functions

`xlsfinfo`	Determine if file contains Microsoft Excel (`.xls`) spreadsheet
`xlsread`	Read Microsoft Excel spreadsheet file (`.xls`)

Lotus123 Functions

`wk1read`	Read Lotus123 WK1 spreadsheet file into matrix
`wk1write`	Write matrix to Lotus123 WK1 spreadsheet file

Scientific Data

Common Data Format (CDF)

`cdfinfo`	Return information about CDF file
`cdfread`	Read CDF file

Flexible Image Transport System

`fitsinfo`	Return information about FITS file
`fitsread`	Read FITS file

Hierarchical Data Format (HDF)

`hdf`	Interface to HDF files
`hdfinfo`	Return information about HDF or HDF-EOS file
`hdfread`	Read HDF file

Audio and Audio/Video

- "General"
- "SPARCstation-Specific Sound Functions"
- "Microsoft WAVE Sound Functions"
- "Audio Video Interleaved (AVI) Functions"
- "Microsoft Excel Functions"
- "Lotus123 Functions"

General

`audioplayer`	Create audio player object
`audiorecorder`	Perform real-time audio capture
`beep`	Produce beep sound
`lin2mu`	Convert linear audio signal to mu-law
`mu2lin`	Convert mu-law audio signal to linear
`sound`	Convert vector into sound
`soundsc`	Scale data and play as sound

SPARCstation-Specific Sound Functions

`auread`	Read NeXT/SUN (`.au`) sound file
`auwrite`	Write NeXT/SUN (`.au`) sound file

Microsoft WAVE Sound Functions

`wavplay`	Play sound on PC-based audio output device
`wavread`	Read Microsoft WAVE (`.wav`) sound file
`wavrecord`	Record sound using PC-based audio input device
`wavwrite`	Write Microsoft WAVE (`.wav`) sound file

Audio Video Interleaved (AVI) Functions

`addframe`	Add frame to AVI file
`avifile`	Create new AVI file
`aviinfo`	Return information about AVI file
`aviread`	Read AVI file
`close`	Close AVI file
`movie2avi`	Create AVI movie from MATLAB movie

Images

`imfinfo`	Return information about graphics file
`imread`	Read image from graphics file
`imwrite`	Write image to graphics file

Graphics

2-D graphs, specialized plots (e.g., pie charts, histograms, and contour plots), function plotters, and Handle Graphics functions.

Category	Description
Basic Plots and Graphs	Linear line plots, log and semilog plots
Annotating Plots	Titles, axes labels, legends, mathematical symbols
Specialized Plotting	Bar graphs, histograms, pie charts, contour plots, function plotters
Bit-Mapped Images	Display image object, read and write graphics file, convert to movie frames
Printing	Printing and exporting figures to standard formats
Handle Graphics	Creating graphics objects, setting properties, finding handles

Basic Plots and Graphs

`box`	Axis box for 2-D and 3-D plots
`errorbar`	Plot graph with error bars
`hold`	Hold current graph
`loglog`	Plot using log-log scales
`polar`	Polar coordinate plot
`plot`	Plot vectors or matrices.
`plot3`	Plot lines and points in 3-D space
`plotyy`	Plot graphs with Y tick labels on the left and right
`semilogx`	Semi-log scale plot
`semilogy`	Semi-log scale plot
`subplot`	Create axes in tiled positions

Annotating Plots

`clabel`	Add contour labels to contour plot
`datetick`	Date formatted tick labels

`gtext`	Place text on 2-D graph using mouse
`legend`	Graph legend for lines and patches
`texlabel`	Produce the TeX format from character string
`title`	Titles for 2-D and 3-D plots
`xlabel`	X-axis labels for 2-D and 3-D plots
`ylabel`	Y-axis labels for 2-D and 3-D plots
`zlabel`	Z-axis labels for 3-D plots

Specialized Plotting

- "Area, Bar, and Pie Plots"
- "Contour Plots"
- "Direction and Velocity Plots"
- "Discrete Data Plots"
- "Function Plots"
- "Histograms"
- "Polygons and Surfaces"
- "Scatter Plots"

Area, Bar, and Pie Plots

`area`	Area plot
`bar`	Vertical bar chart
`barh`	Horizontal bar chart
`bar3`	Vertical 3-D bar chart
`bar3h`	Horizontal 3-D bar chart
`pareto`	Pareto char
`pie`	Pie plot
`pie3`	3-D pie plot

Contour Plots

`contour`	Contour (level curves) plot
`contourc`	Contour computation
`contourf`	Filled contour plot
`ezcontour`	Easy to use contour plotter
`ezcontourf`	Easy to use filled contour plotter

Direction and Velocity Plots

`comet`	Comet plot
`comet3`	3-D comet plot

`compass`	Compass plot
`feather`	Feather plot
`quiver`	Quiver (or velocity) plot
`quiver3`	3-D quiver (or velocity) plot

Discrete Data Plots

`stem`	Plot discrete sequence data
`stem3`	Plot discrete surface data
`stairs`	Stairstep graph

Function Plots

`ezcontour`	Easy to use contour plotter
`ezcontourf`	Easy to use filled contour plotter
`ezmesh`	Easy to use 3-D mesh plotter
`ezmeshc`	Easy to use combination mesh/contour plotter
`ezplot`	Easy to use function plotter
`ezplot3`	Easy to use 3-D parametric curve plotter
`ezpolar`	Easy to use polar coordinate plotter
`ezsurf`	Easy to use 3-D colored surface plotter
`ezsurfc`	Easy to use combination surface/contour plotter
`fplot`	Plot a function

Histograms

`hist`	Plot histograms
`histc`	Histogram count
`rose`	Plot rose or angle histogram

Polygons and Surfaces

`convhull`	Convex hull
`cylinder`	Generate cylinder
`delaunay`	Delaunay triangulation
`dsearch`	Search Delaunay triangulation for nearest point
`ellipsoid`	Generate ellipsoid
`fill`	Draw filled 2-D polygons
`fill3`	Draw filled 3-D polygons in 3-space
`inpolygon`	True for points inside a polygonal region
`pcolor`	Pseudocolor (checkerboard) plot
`polyarea`	Area of polygon
`ribbon`	Ribbon plot
`slice`	Volumetric slice plot
`sphere`	Generate sphere

`tsearch`	Search for enclosing Delaunay triangle
`voronoi`	Voronoi diagram
`waterfall`	Waterfall plot

Scatter Plots

`plotmatrix`	Scatter plot matrix
`scatter`	Scatter plot
`scatter3`	3-D scatter plot

Bit-Mapped Images

`frame2im`	Convert movie frame to indexed image
`image`	Display image object
`imagesc`	Scale data and display image object
`imfinfo`	Information about graphics file
`im2frame`	Convert image to movie frame
`imread`	Read image from graphics file
`imwrite`	Write image to graphics file
`ind2rgb`	Convert indexed image to RGB image

Printing

`orient`	Hardcopy paper orientation
`pagesetupdlg`	Page position dialog box
`print`	Print graph or save graph to file
`printdlg`	Print dialog box
`printopt`	Configure local printer defaults
`printpreview`	Preview figure to be printed
`saveas`	Save figure to graphic file

Handle Graphics

- Finding and Identifying Graphics Objects
- Object Creation Functions
- Figure Windows
- Axes Operations

Finding and Identifying Graphics Objects

`allchild`	Find all children of specified objects
`copyobj`	Make copy of graphics object and its children

`delete`	Delete files or graphics objects
`findall`	Find all graphics objects (including hidden handles)
`findobj`	Find objects with specified property values
`gca`	Get current Axes handle
`gcbo`	Return object whose callback is currently executing
`gcbf`	Return handle of figure containing callback object
`gco`	Return handle of current object
`get`	Get object properties
`ishandle`	True if value is valid object handle
`rotate`	Rotate objects about specified origin and direction
`set`	Set object properties

Object Creation Functions

`axes`	Create axes object
`figure`	Create figure (graph) windows
`image`	Create image (2-D matrix)
`light`	Create light object (illuminates Patch and Surface)
`line`	Create line object (3-D polylines)
`patch`	Create patch object (polygons)
`rectangle`	Create rectangle object (2-D rectangle)
`surface`	Create surface (quadrilaterals)
`text`	Create text object (character strings)
`uicontextmenu`	Create context menu (popup associated with object)

Figure Windows

`capture`	Screen capture of the current figure
`clc`	Clear figure window
`clf`	Clear figure
`close`	Close specified window
`closereq`	Default close request function
`drawnow`	Complete any pending drawing
`gcf`	Get current figure handle
`newplot`	Graphics M-file preamble for `NextPlot` property
`refresh`	Refresh figure
`saveas`	Save figure or model to desired output format

Axes Operations

`axis`	Plot axis scaling and appearance
`cla`	Clear Axes
`gca`	Get current Axes handle
`grid`	Grid lines for 2-D and 3-D plots

3-D Visualization

Create and manipulate graphics that display 2-D matrix and 3-D volume data, controlling the view, lighting and transparency.

Category	Description
Surface and Mesh Plots	Plot matrices, visualize functions of two variables, specify colormap
View Control	Control the camera viewpoint, zooming, rotation, aspect ratio, set axis limits
Lighting	Add and control scene lighting
Transparency	Specify and control object transparency
Volume Visualization	Visualize gridded volume data

Surface and Mesh Plots

- Creating Surfaces and Meshes
- Domain Generation
- Color Operations
- Colormaps

Creating Surfaces and Meshes

`hidden`	Mesh hidden line removal mode
`meshc`	Combination mesh/contourplot
`mesh`	3-D mesh with reference plane
`peaks`	A sample function of two variables
`surf`	3-D shaded surface graph
`surface`	Create surface low-level objects
`surfc`	Combination surf/contourplot
`surfl`	3-D shaded surface with lighting
`tetramesh`	Tetrahedron mesh plot
`trimesh`	Triangular mesh plot
`triplot`	2-D triangular plot
`trisurf`	Triangular surface plot

Domain Generation

`griddata`	Data gridding and surface fitting
`meshgrid`	Generation of X and Y arrays for 3-D plots

Color Operations

`brighten`	Brighten or darken color map
`caxis`	Pseudocolor axis scaling
`colorbar`	Display color bar (color scale)
`colordef`	Set up color defaults
`colormap`	Set the color look-up table (list of colormaps)
`graymon`	Graphics figure defaults set for grayscale monitor
`hsv2rgb`	Hue-saturation-value to red-green-blue conversion
`rgb2hsv`	RGB to HSVconversion
`rgbplot`	Plot color map
`shading`	Color shading mode
`spinmap`	Spin the colormap
`surfnorm`	3-D surface normals
`whitebg`	Change axes background color for plots

Colormaps

`autumn`	Shades of red and yellow color map
`bone`	Gray-scale with a tinge of blue color map
`contrast`	Gray color map to enhance image contrast
`cool`	Shades of cyan and magenta color map
`copper`	Linear copper-tone color map
`flag`	Alternating red, white, blue, and black color map
`gray`	Linear gray-scale color map
`hot`	Black-red-yellow-white color map
`hsv`	Hue-saturation-value (HSV) color map
`jet`	Variant of HSV
`lines`	Line color colormap
`prism`	Colormap of prism colors
`spring`	Shades of magenta and yellow color map
`summer`	Shades of green and yellow colormap
`winter`	Shades of blue and green color map

View Control

- Controlling the Camera Viewpoint
- Setting the Aspect Ratio and Axis Limits
- Object Manipulation

- Selecting Region of Interest

Controlling the Camera Viewpoint

`camdolly`	Move camera position and target
`camlookat`	View specific objects
`camorbit`	Orbit about camera target
`campan`	Rotate camera target about camera position
`campos`	Set or get camera position
`camproj`	Set or get projection type
`camroll`	Rotate camera about viewing axis
`camtarget`	Set or get camera target
`camup`	Set or get camera up-vector
`camva`	Set or get camera view angle
`camzoom`	Zoom camera in or out
`view`	3-D graph viewpoint specification.
`viewmtx`	Generate view transformation matrices

Setting the Aspect Ratio and Axis Limits

`daspect`	Set or get data aspect ratio
`pbaspect`	Set or get plot box aspect ratio
`xlim`	Set or get the current *x*-axis limits
`ylim`	Set or get the current *y*-axis limits
`zlim`	Set or get the current *z*-axis limits

Object Manipulation

`reset`	Reset axis or figure
`rotate3d`	Interactively rotate the view of a 3-D plot
`selectmoveresize`	Interactively select, move, or resize objects
`zoom`	Zoom in and out on a 2-D plot

Selecting Region of Interest

`dragrect`	Drag XOR rectangles with mouse
`rbbox`	Rubberband box

Lighting

`camlight`	Cerate or position Light
`light`	Light object creation function
`lightangle`	Position light in sphereical coordinates
`lighting`	Lighting mode
`material`	Material reflectance mode

Transparency

`alpha`	Set or query transparency properties for objects in current axes
`alphamap`	Specify the figure alphamap
`alim`	Set or query the axes alpha limits

Volume Visualization

`coneplot`	Plot velocity vectors as cones in 3-D vector field
`contourslice`	Draw contours in volume slice plane
`curl`	Compute curl and angular velocity of vector field
`divergence`	Compute divergence of vector field
`flow`	Generate scalar volume data
`interpstreamspeed`	Interpolate streamline vertices from vector-field magnitudes
`isocaps`	Compute isosurface end-cap geometry
`isocolors`	Compute colors of isosurface vertices
`isonormals`	Compute normals of isosurface vertices
`isosurface`	Extract isosurface data from volume data
`reducepatch`	Reduce number of patch faces
`reducevolume`	Reduce number of elements in volume data set
`shrinkfaces`	Reduce size of patch faces
`slice`	Draw slice planes in volume
`smooth3`	Smooth 3-D data
`stream2`	Compute 2-D stream line data
`stream3`	Compute 3-D stream line data
`streamline`	Draw stream lines from 2- or 3-D vector data
`streamparticles`	Draws stream particles from vector volume data
`streamribbon`	Draws stream ribbons from vector volume data
`streamslice`	Draws well-spaced stream lines from vector volume data
`streamtube`	Draws stream tubes from vector volume data
`surf2patch`	Convert surface data to patch data
`subvolume`	Extract subset of volume data set
`volumebounds`	Return coordinate and color limits for volume (scalar and vector)

Creating Graphical User Interfaces

Predefined dialog boxes and functions to control GUI programs.

Category	Description
Predefined Dialog Boxes	Dialog boxes for error, user input, waiting, etc.
Deploying User Interfaces	Launching GUIs, creating the handles structure
Developing User Interfaces	Starting GUIDE, managing application data, getting user input
User Interface Objects	Creating GUI components
Finding and Identifying Objects	Finding object handles from callbacks
GUI Utility Functions	Moving objects, text wrapping
Controlling Program Execution	Wait and resume based on user input

Predefined Dialog Boxes

`dialog`	Create dialog box
`errordlg`	Create error dialog box
`helpdlg`	Display help dialog box
`inputdlg`	Create input dialog box
`listdlg`	Create list selection dialog box
`msgbox`	Create message dialog box
`pagedlg`	Display page layout dialog box
`printdlg`	Display print dialog box
`questdlg`	Create question dialog box
`uigetfile`	Display dialog box to retrieve name of file for reading
`uiputfile`	Display dialog box to retrieve name of file for writing
`uisetcolor`	Set `ColorSpec` using dialog box
`uisetfont`	Set font using dialog box
`waitbar`	Display wait bar
`warndlg`	Create warning dialog box

Deploying User Interfaces

`guidata`	Store or retrieve application data
`guihandles`	Create a structure of handles
`movegui`	Move GUI figure onscreen
`openfig`	Open or raise GUI figure

Developing User Interfaces

`guide`	Open GUI Layout Editor
`inspect`	Display Property Inspector

Working with Application Data

`getappdata`	Get value of application data
`isappdata`	True if application data exists
`rmappdata`	Remove application data
`setappdata`	Specify application data

Interactive User Input

`ginput`	Graphical input from a mouse or cursor
`waitforbuttonpress`	Wait for key/buttonpress over figure

User Interface Objects

`menu`	Generate menu of choices for user input
`uicontextmenu`	Create context menu
`uicontrol`	Create user interface control
`uimenu`	Create user interface menu

Finding and Identifying Objects

`findall`	Find all graphics objects
`findfigs`	Display off-screen visible figure windows
`gcbf`	Return handle of figure containing callback object
`gcbo`	Return handle of object whose callback is executing

GUI Utility Functions

`selectmoveresize`	Select, move, resize, or copy axes and uicontrol graphics objects
`textwrap`	Return wrapped string matrix for given uicontrol

Controlling Program Execution

`uiresume`	Resumes program execution halted with `uiwait`
`uiwait`	Halts program execution, restart with `uiresume`

2

Alphabetical List of Functions

Purpose Consolidate workspace memory

Syntax

```
pack
pack filename
pack('filename')
```

Description `pack` frees up needed space by compressing information into the minimum memory required. You must run `pack` from a directory for which you have write permission.

`pack filename` accepts an optional `filename` for the temporary file used to hold the variables. Otherwise, it uses the file named `pack.tmp`. You must run `pack` from a directory for which you have write permission.

`pack('filename')` is the function form of `pack`.

Remarks The `pack` function does not affect the amount of memory allocated to the MATLAB process. You must quit MATLAB to free up this memory.

Since MATLAB uses a heap method of memory management, extended MATLAB sessions may cause memory to become fragmented. When memory is fragmented, there may be plenty of free space, but not enough contiguous memory to store a new large variable.

If you get the `Out of memory` message from MATLAB, the `pack` function may find you some free memory without forcing you to delete variables.

The `pack` function frees space by:

- Saving all variables on disk in a temporary file called `pack.tmp`
- Clearing all variables and functions from memory
- Reloading the variables back from `pack.tmp`
- Deleting the temporary file `pack.tmp`

If you use `pack` and there is still not enough free memory to proceed, you must clear some variables. If you run out of memory often, you can allocate larger matrices earlier in the MATLAB session and use these system-specific tips:

- UNIX: Ask your system manager to increase your swap space.
- Windows: Increase virtual memory using the Windows Control Panel.

Examples

Change the current directory to one that is writable, run pack, and return to the previous directory.

```
cwd = pwd;
cd(tempdir);
pack
cd(cwd)
```

See Also

clear

Purpose

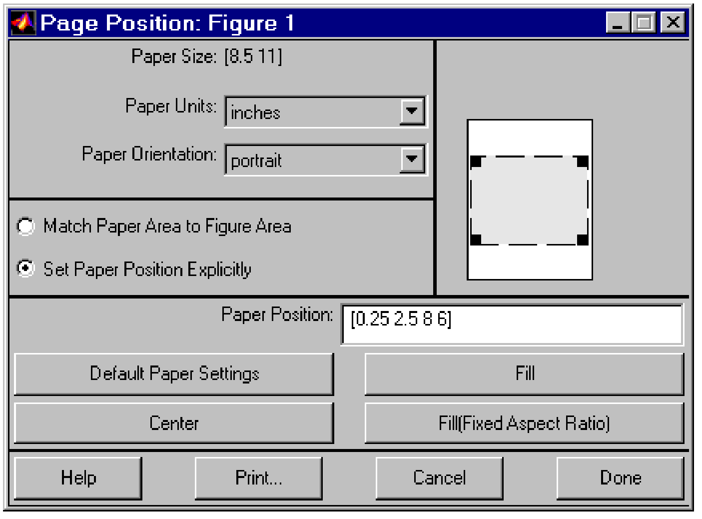
This function is obsolete. Use `pagesetupdlg` to display the page setup dialog.

Syntax

```
pagedlg
pagedlg(fig)
```

Description

`pagedlg` displays a page position dialog box for the current figure. The dialog box enables you to set page layout properties.



`pagedlg(fig)` displays a page position dialog box for the figure identified by the handle `fig`.

Remarks

This dialog box enables you to set figure properties that determine how MATLAB lays out the figure on the printed paper. See the dialog box help for more information.

See Also

The figure properties – `PaperPosition`, `PaperOrientation`, `PaperUnits`

pagesetupdlg

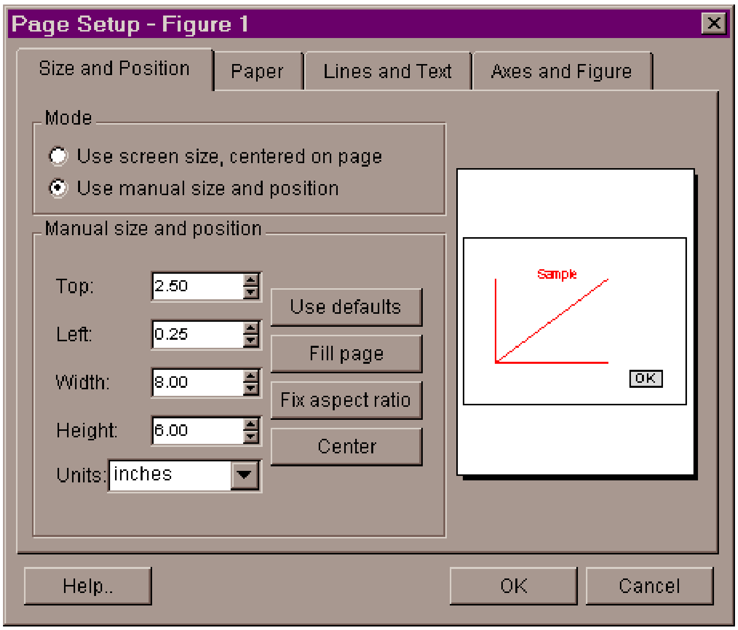
Purpose Page position dialog box

Syntax `dlg = pagesetupdlg(fig)`

Description `dlg = pagesetupdlg(fig)` creates a dialog box from which a set of pagelayout properties for the figure window, `fig`, can be set.

`pagesetupdlg` implements the "Page Setup..." option in the **Figure File Menu**.

Unlike `pagedlg`, `pagesetupdlg` currently only supports setting the layout for a single figure. `fig` must be a single figure handle, not a vector of figures or a simulink diagram.



See Also `pagedlg`, `printpreview`, `printopt`

Purpose Pareto chart

Syntax

```
pareto(Y)
pareto(Y,names)
pareto(Y,X)
H = pareto(...)
```

Description Pareto charts display the values in the vector `Y` as bars drawn in descending order.

`pareto(Y)` labels each bar with its element index in `Y`.

`pareto(Y,names)` labels each bar with the associated name in the string matrix or cell array `names`.

`pareto(Y,X)` labels each bar with the associated value from `X`.

`H = pareto(...)` returns a combination of patch and line object handles.

See Also `hist`, `bar`

partialpath

Purpose

Partial pathname

Description

A partial pathname is a pathname relative to the MATLAB path, `MATLABPATH`. It is used to locate private and method files, which are usually hidden, or to restrict the search for files when more than one file with the given name exists.

A partial pathname contains the last component, or last several components, of the full pathname separated by `/`. For example, `matfun/trace`, `private/children`, `inline/formula`, and `demos/clown.mat` are valid partial pathnames. Specifying the `@` in method directory names is optional, so `funfun/inline/formula` is also a valid partial pathname.

Partial pathnames make it easy to find toolbox or MATLAB relative files on your path in a portable way, independent of the location where MATLAB is installed.

Many commands accept partial pathnames instead of a full pathname. Some of these commands are

```
help, type, load, exist, what, which, edit, dbtype, dbstop,
dbclear, and fopen
```

Examples

The following examples use partial pathnames.

```
what funfun/inline

M-files in directory matlabroot\toolbox\matlab\funfun\@inline
argnames   disp      feval     inline     subsref    vertcat
cat        display    formula    nargin     symvar
char       exist      horzcat    nargout    vectorize
```

```
which funfun/inline/formula
matlabroot\toolbox\matlab\funfun\@inline\formula.m
% inline method
```

See Also

`path`

Purpose Pascal matrix

Syntax

```
A = pascal(n)
A = pascal(n,1)
A = pascal(n,2)
```

Description

`A = pascal(n)` returns the Pascal matrix of order `n`: a symmetric positive definite matrix with integer entries taken from Pascal's triangle. The inverse of `A` has integer entries.

`A = pascal(n,1)` returns the lower triangular Cholesky factor (up to the signs of the columns) of the Pascal matrix. It is *involutary*, that is, it is its own inverse.

`A = pascal(n,2)` returns a transposed and permuted version of `pascal(n,1)`. `A` is a cube root of the identity matrix.

Examples

`pascal(4)` returns

```
1     1     1     1
1     2     3     4
1     3     6    10
1     4    10    20
```

`A = pascal(3,2)` produces

```
A =
     0     0    -1
     0    -1     2
    -1    -1     1
```

See Also

`chol`

Purpose Create patch graphics object

Syntax

```
patch(X,Y,C)
patch(X,Y,Z,C)
patch(FV)
patch(...'PropertyName',PropertyValue...)
patch('PropertyName',PropertyValue...)  PN/PV pairs only
handle = patch(...)
```

Description

`patch` is the low-level graphics function for creating patch graphics objects. A patch object is one or more polygons defined by the coordinates of its vertices. You can specify the coloring and lighting of the patch. See the Creating 3-D Models with Patches for more information on using patch objects.

`patch(X,Y,C)` adds the filled two-dimensional patch to the current axes. The elements of `X` and `Y` specify the vertices of a polygon. If `X` and `Y` are matrices, MATLAB draws one polygon per column. `C` determines the color of the patch. It can be a single `ColorSpec`, one color per face, or one color per vertex (see "Remarks"). If C is a 1-by-3 vector, it is assumed to be an RGB triplet, specifying a color directly.

`patch(X,Y,Z,C)` creates a patch in three-dimensional coordinates.

`patch(FV)` creates a patch using structure `FV`, which contains the fields `vertices`, `faces`, and optionally `facevertecdata`. These fields correspond to the `Vertices`, `Faces`, and `FaceVertexCData` patch properties.

`patch(...'`*`PropertyName`*`',PropertyValue...)` follows the `X`, `Y`, (`Z`), and `C` arguments with property name/property value pairs to specify additional patch properties.

`patch('`*`PropertyName`*`',PropertyValue,...)` specifies all properties using property name/property value pairs. This form enables you to omit the color specification because MATLAB uses the default face color and edge color, unless you explicitly assign a value to the `FaceColor` and `EdgeColor` properties. This form also allows you to specify the patch using the `Faces` and `Vertices` properties instead of *x*-, *y*-, and *z*-coordinates. See the "Examples" section for more information.

handle = patch(...) returns the handle of the patch object it creates.

Remarks

Unlike high-level area creation functions, such as fill or area, patch does not check the settings of the figure and axes NextPlot properties. It simply adds the patch object to the current axes.

If the coordinate data does not define closed polygons, patch closes the polygons. The data can define concave or intersecting polygons. However, if the edges of an individual patch face intersect themselves, the resulting face may or may not be completely filled. In that case, it is better to break up the face into smaller polygons.

Specifying Patch Properties

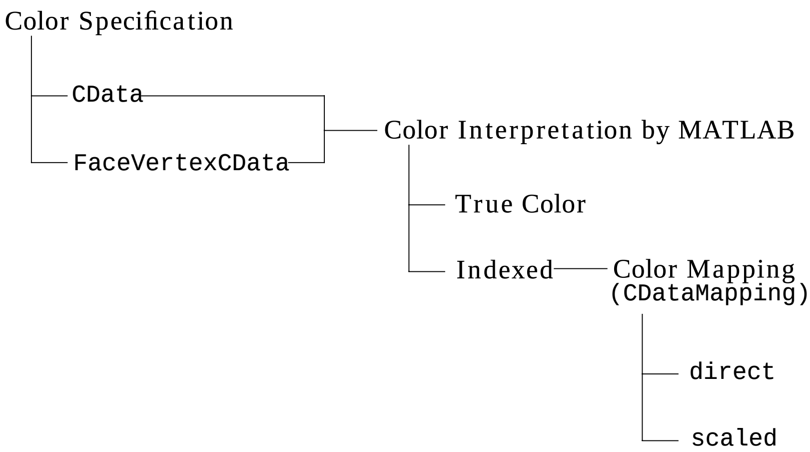
You can specify properties as property name/property value pairs, structure arrays, and cell arrays (see the set and get reference pages for examples of how to specify these data types).

There are two patch properties that specify color:

- CData – use when specifying *x*-, *y*-, and *z*-coordinates (XData, YData, ZData).
- FaceVertexCData – use when specifying vertices and connection matrix (Vertices and Faces).

The CData and FaceVertexCData properties accept color data as indexed or true color (RGB) values. See the CData and FaceVertexCData property descriptions for information on how to specify color.

Indexed color data can represent either direct indices into the colormap or scaled values that map the data linearly to the entire colormap (see the caxis function for more information on this scaling). The `CDataMapping` property determines how MATLAB interprets indexed color data.



Color Data Interpretation

You can specify patch colors as:

- A single color for all faces
- One color for each face enabling flat coloring
- One color for each vertex enabling interpolated coloring

The following tables summarize how MATLAB interprets color data defined by the `CData` and `FaceVertexCData` properties.

Interpretation of the CData Property

[X,Y,Z]Data Dimensions	CData Required for Indexed	CData Required for True Color	Results Obtained
m-by-n	scalar	1-by-1-by-3	Use the single color specified for all patch faces. Edges can be only a single color.

[X,Y,Z]Data Dimensions	CData Required for Indexed	True Color	Results Obtained
m-by-n	1-by-n (n >= 4)	1-by-n-by-3	Use one color for each patch face. Edges can be only a single color.
m-by-n	m-by-n	m-by-n-3	Assign a color to each vertex. patch faces can be flat (a single color) or interpolated. Edges can be flat or interpolated.

Interpretation of the FaceVertexCData Property

Vertices Dimensions	Faces Dimensions	FaceVertexCData Required for Indexed	True Color	Results Obtained
m-by-n	k-by-3	scalar	1-by-3	Use the single color specified for all patch faces. Edges can be only a single color.
m-by-n	k-by-3	k-by-1	k-by-3	Use one color for each patch face. Edges can be only a single color.
m-by-n	k-by-3	m-by-1	m-by-3	Assign a color to each vertex. patch faces can be flat (a single color) or interpolated. Edges can be flat or interpolated.

Examples

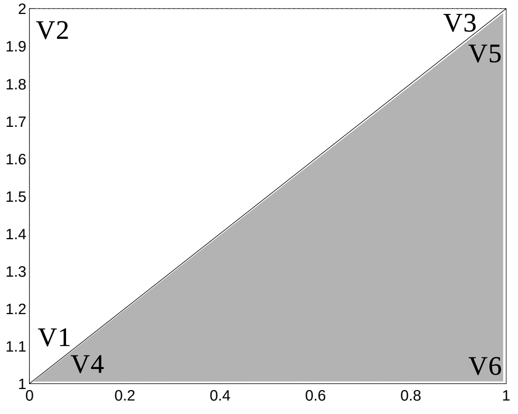
This example creates a patch object using two different methods:

- Specifying *x*-, *y*-, and *z*-coordinates and color data (`XData`, `YData`, `ZData`, and `CData` properties).
- Specifying vertices, the connection matrix, and color data (`Vertices`, `Faces`, `FaceVertexCData`, and `FaceColor` properties).

Specifying X, Y, and Z Coordinates

The first approach specifies the coordinates of each vertex. In this example, the coordinate data defines two triangular faces, each having three vertices. Using true color, the top face is set to white and the bottom face to gray.

```
x = [0 0;0 1;1 1];
y = [1 1;2 2;2 1];
z = [1 1;1 1;1 1];
tcolor(1,1,1:3) = [1 1 1];
tcolor(1,2,1:3) = [.7 .7 .7];
patch(x,y,z,tcolor)
```



Notice that each face shares two vertices with the other face (V_1-V_4 and V_3-V_5).

Specifying Vertices and Faces

The `Vertices` property contains the coordinates of each *unique* vertex defining the patch. The `Faces` property specifies how to connect these vertices to form each face of the patch. For this example, two vertices share the same location so you need to specify only four of the six vertices. Each row contains the *x*, *y*, and *z*-coordinates of each vertex.

```
vert = [0 1 1;0 2 1;1 2 1;1 1 1];
```

There are only two faces, defined by connecting the vertices in the order indicated.

```
fac = [1 2 3;1 3 4];
```

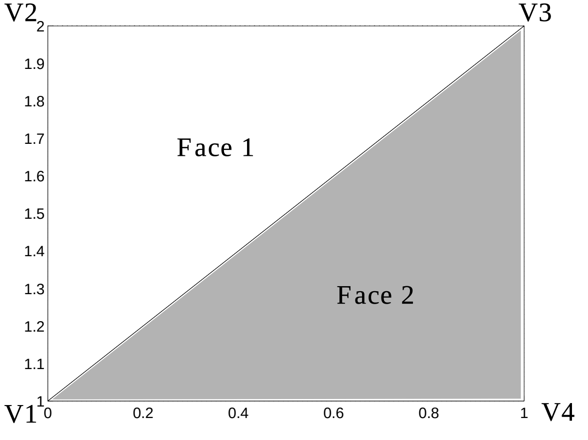
To specify the face colors, define a 2-by-3 matrix containing two RGB color definitions.

```
tcolor = [1 1 1;.7 .7 .7];
```

With two faces and two colors, MATLAB can color each face with flat shading. This means you must set the `FaceColor` property to `flat`, since the faces/vertices technique is available only as a low-level function call (i.e., only by specifying property name/property value pairs).

Create the patch by specifying the `Faces`, `Vertices`, and `FaceVertexCData` properties as well as the `FaceColor` property.

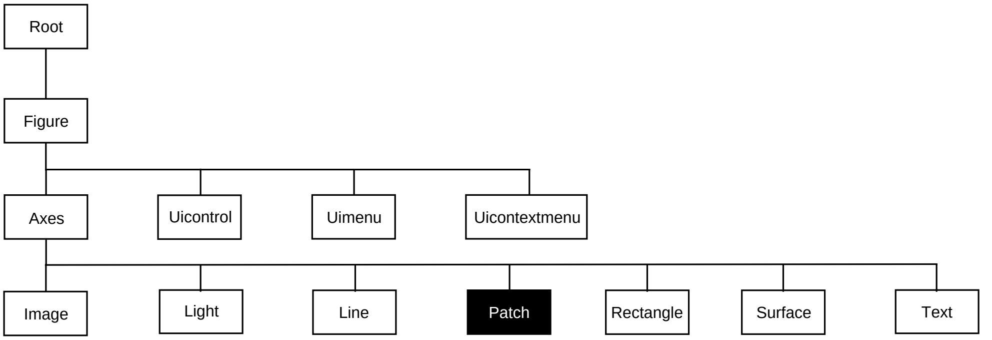
```
patch('Faces',fac,'Vertices',vert,'FaceVertexCData',tcolor,...
      'FaceColor','flat')
```



Specifying only unique vertices and their connection matrix can reduce the size of the data for patches having many faces. See the descriptions of the `Faces`, `Vertices`, and `FaceVertexCData` properties for information on how to define them.

MATLAB does not require each face to have the same number of vertices. In cases where they do not, pad the `Faces` matrix with `NaN`s. To define a patch with faces that do not close, add one or more `NaN` to the row in the `Vertices` matrix that defines the vertex you do not want connected.

Object Hierarchy



Setting Default Properties

You can set default patch properties on the axes, figure, and root levels.

```
set(0,'DefaultPatchPropertyName',PropertyValue...)
set(gcf,'DefaultPatchPropertyName',PropertyValue...)
set(gca,'DefaultPatchPropertyName',PropertyValue...)
```

`PropertyName` is the name of the patch property and `PropertyValue` is the value you are specifying. Use `set` and `get` to access patch properties.

Property List

The following table lists all patch properties and provides a brief description of each. The property name links take you to an expanded description of the properties.

Property Name	Property Description	Property Value
Data Defining the Object		
`Faces`	Connection matrix for `Vertices`	Values: m-by-n matrix Default: `[1,2,3]`

Property Name	Property Description	Property Value
Vertices	Matrix of *x*-, *y*-, and *z*-coordinates of the vertices (used with Faces)	Values: matrix Default: [0,1;1,1;0,0]
XData	The *x*-coordinates of the vertices of the patch	Values: vector or matrix Default: [0;1;0]
YData	The *y*-coordinates of the vertices of the patch	Values: vector or matrix Default: [1;1;0]
ZData	The *z*-coordinates of the vertices of the patch	Values: vector or matrix Default: [] empty matrix
Specifying Color		
CData	Color data for use with the XData/YData/ZData method	Values: scalar, vector, or matrix Default: [] empty matrix
CDataMapping	Controls mapping of CData to colormap	Values: scaled, direct Default: scaled
EdgeColor	Color of face edges	Values: ColorSpec, none, flat, interp Default: ColorSpec
FaceColor	Color of face	Values: ColorSpec, none, flat, interp Default: ColorSpec
FaceVertexCData	Color data for use with Faces/Vertices method	Values: matrix Default: [] empty matrix
MarkerEdgeColor	Color of marker or the edge color for filled markers	Values: ColorSpec, none, auto Default: auto
MarkerFaceColor	Fill color for markers that are closed shapes	Values: ColorSpec, none, auto Default: none
Controlling the Effects of Lights		

Property Name	Property Description	Property Value
`AmbientStrength`	Intensity of the ambient light	Values: scalar >=0 and <=1 Default: 0.3
`BackFaceLighting`	Controls lighting of faces pointing away from camera	Values: unlit, lit, reverselit Default: reverselit
`DiffuseStrength`	Intensity of diffuse light	Values: scalar >=0 and <=1 Default: 0.6
`EdgeLighting`	Method used to light edges	Values: none, flat, gouraud, phong Default: none
`FaceLighting`	Method used to light edges	Values: none, flat, gouraud, phong Default: none
NormalMode	MATLAB-generated or user-specified normal vectors	Values: auto, manual Default: auto
`SpecularColorReflectance`	Composite color of specularly reflected light	Values: scalar 0 to 1 Default: 1
`SpecularExponent`	Harshness of specular reflection	Values: scalar >= 1 Default: 10
`SpecularStrength`	Intensity of specular light	Values: scalar >=0 and <=1 Default: 0.9
VertexNormals	Vertex normal vectors	Values: matrix
Defining Edges and Markers		
`LineStyle`	Select from five line styles.	Values: –, ––, :, –., none Default: –
`LineWidth`	The width of the edge in points	Values: scalar Default: 0.5 points

Property Name	Property Description	Property Value
Marker	Marker symbol to plot at data points	Values: see Marker property Default: none
MarkerSize	Size of marker in points	Values: size in points Default: 6
Specifying Transparency		
AlphaDataMapping	Transparency mapping method	none, direct, scaled Default: scaled
EdgeAlpha	Transparency of the edges of patch faces	scalar, flat, interp Default: 1 (opaque)
FaceAlpha	Transparency of the patch face	scalar, flat, interp Default: 1 (opaque)
FaceVertexAlphaData	Face and vertex transparency data	m-by-1 matrix
Controlling the Appearance		
Clipping	Clipping to axes rectangle	Values: on, off Default: on
EraseMode	Method of drawing and erasing the patch (useful for animation)	Values: normal, none, xor, background Default: normal
SelectionHighlight	Highlight patch when selected (Selected property set to on)	Values: on, off Default: on
Visible	Make the patch visible or invisible	Values: on, off Default: on
Controlling Access to Objects		
HandleVisibility	Determines if and when the the patch's handle is visible to other functions	Values: on, callback, off Default: on

Property Name	Property Description	Property Value
HitTest	Determines if the patch can become the current object (see the figure CurrentObject property)	Values: on, off Default: on
Controlling Callback Routine Execution		
BusyAction	Specify how to handle callback routine interruption	Values: cancel, queue Default: queue
ButtonDownFcn	Define a callback routine that executes when a mouse button is pressed on over the patch	Values: string Default: '' (empty string)
CreateFcn	Define a callback routine that executes when an patch is created	Values: string Default: '' (empty string)
DeleteFcn	Define a callback routine that executes when the patch is deleted (via close or delete)	Values: string Default: '' (empty string)
Interruptible	Determine if callback routine can be interrupted	Values: on, off Default: on (can be interrupted)
UIContextMenu	Associate a context menu with the patch	Values: handle of a Uicontrextmenu
General Information About the Patch		
Children	Patch objects have no children	Values: [] (empty matrix)
Parent	The parent of a patch object is always an axes object	Value: axes handle
Selected	Indicate whether the patch is in a "selected" state.	Values: on, off Default: on
Tag	User-specified label	Value: any string Default: '' (empty string)

Property Name	Property Description	Property Value
`Type`	The type of graphics object (read only)	Value: the string `'patch'`
`UserData`	User-specified data	Values: any matrix Default: `[]` (empty matrix)

See Also `area`, `caxis`, `fill`, `fill3`, `isosurface`, `surface`

Patch Properties

Modifying Properties

You can set and query graphics object properties in two ways:

- The Property Editor is an interactive tool that enables you to see and change object property values.
- The `set` and `get` commands enable you to set and query the values of properties

To change the default value of properties see Setting Default Property Values.

Patch Property Descriptions

This section lists property names along with the type of values each accepts. Curly braces { } enclose default values.

`AlphaDataMapping` `none | direct | {scaled}`

Transparency mapping method. This property determines how MATLAB interprets indexed alpha data. This property can be any of the following:

- `none` - The transparency values of `FaceVertexAlphaData` are between 0 and 1 or are clamped to this range (the default).
- `scaled` - Transform the `FaceVertexAlphaData` to span the portion of the alphamap indicated by the axes `ALim` property, linearly mapping data values to alpha values.
- `direct` - use the `FaceVertexAlphaData` as indices directly into the alphamap. When not scaled, the data are usually integer values ranging from `1` to `length(alphamap)`. MATLAB maps values less than `1` to the first alpha value in the alphamap, and values greater than `length(alphamap)` to the last alpha value in the alphamap. Values with a decimal portion are fixed to the nearest, lower integer. If `FaceVertexAlphaData` is an array `unit8` integers, then the indexing begins at `0` (i.e., MATLAB maps a value of `0` to the first alpha value in the alphamap).

`AmbientStrength` scalar >= 0 and <= 1

Strength of ambient light. This property sets the strength of the ambient light, which is a nondirectional light source that illuminates the entire scene. You must have at least one visible light object in the axes for the ambient light to be visible. The axes `AmbientColor` property sets the color of the ambient light, which is therefore the same on all objects in the axes.

You can also set the strength of the diffuse and specular contribution of light objects. See the `DiffuseStrength` and `SpecularStrength` properties.

BackFaceLighting `unlit | lit | {reverselit}`

Face lighting control. This property determines how faces are lit when their vertex normals point away from the camera:

- `unlit` – face is not lit
- `lit` – face lit in normal way
- `reverselit` – face is lit as if the vertex pointed towards the camera

This property is useful for discriminating between the internal and external surfaces of an object. See the *Using MATLAB Graphics* manual for an example.

BusyAction `cancel | {queue}`

Callback routine interruption. The `BusyAction` property enables you to control how MATLAB handles events that potentially interrupt executing callback routines. If there is a callback routine executing, subsequently invoked callback routes always attempt to interrupt it. If the `Interruptible` property of the object whose callback is executing is set to `on` (the default), then interruption occurs at the next point where the event queue is processed. If the `Interruptible` property is `off`, the `BusyAction` property (of the object owning the executing callback) determines how MATLAB handles the event. The choices are:

- `cancel` – discard the event that attempted to execute a second callback routine.
- `queue` – queue the event that attempted to execute a second callback routine until the current callback finishes.

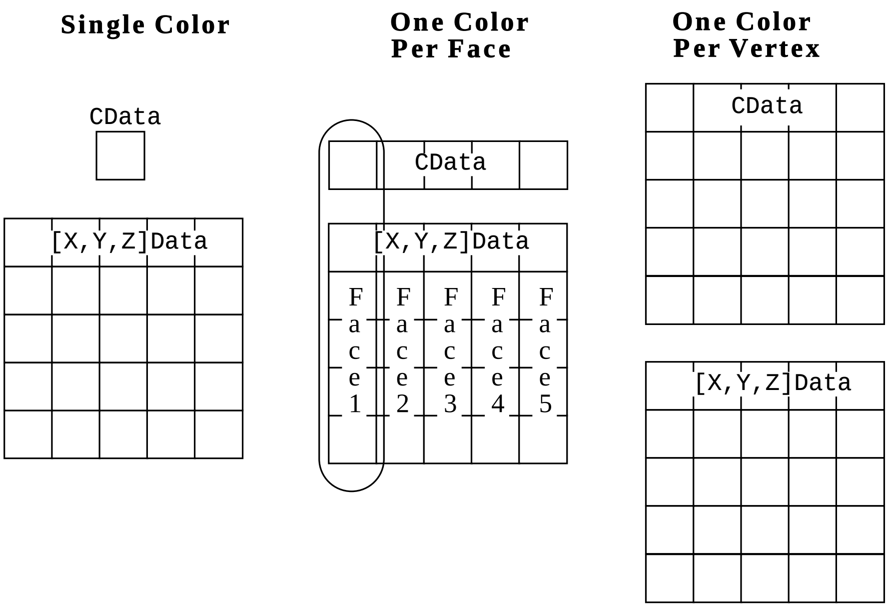
ButtonDownFcn string

Button press callback routine. A callback routine that executes whenever you press a mouse button while the pointer is over the patch object. Define this routine as a string that is a valid MATLAB expression or the name of an M-file. The expression executes in the MATLAB workspace.

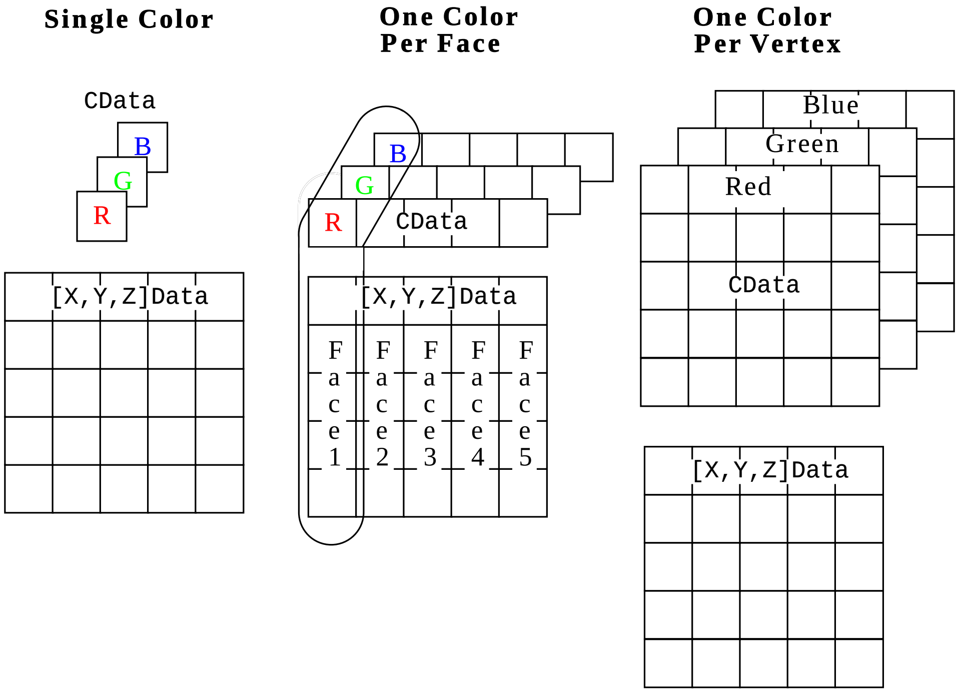
CData scalar, vector, or matrix

Patch colors. This property specifies the color of the patch. You can specify color for each vertex, each face, or a single color for the entire patch. The way MATLAB interprets `CData` depends on the type of data supplied. The data can be numeric values that are scaled to map linearly into the current colormap, integer values that are used directly as indices into the current colormap, or arrays of RGB values. RGB values are not mapped into the current colormap, but interpreted as the colors defined. On true color systems, MATLAB uses the actual colors defined by the RGB triples. On pseudocolor systems, MATLAB uses dithering to approximate the RGB triples using the colors in the figure's `Colormap` and `Dithermap`.

The following two diagrams illustrate the dimensions of `CData` with respect to the coordinate data arrays, `XData`, `YData`, and `ZData`. The first diagram illustrates the use of indexed color.



The second diagram illustrates the use of true color. True color requires *m*-by-*n*-by-3 arrays to define red, green, and blue components for each color.



Note that if `CData` contains `NaNs`, MATLAB does not color the faces.

See also the `Faces`, `Vertices`, and `FaceVertexCData` properties for an alternative method of patch definition.

`CDataMapping` `{scaled} | direct`

Direct or scaled color mapping. This property determines how MATLAB interprets indexed color data used to color the patch. (If you use true color specification for `CData` or `FaceVertexCData`, this property has no effect.)

- `scaled` – transform the color data to span the portion of the colormap indicated by the axes `CLim` property, linearly mapping data values to colors. See the `caxis` command for more information on this mapping.
- `direct` – use the color data as indices directly into the colormap. When not scaled, the data are usually integer values ranging from 1 to

`length(colormap)`. MATLAB maps values less than 1 to the first color in the colormap, and values greater than `length(colormap)` to the last color in the colormap. Values with a decimal portion are fixed to the nearest, lower integer.

`Children` matrix of handles

Always the empty matrix; patch objects have no children.

`Clipping` `{on} | off`

Clipping to axes rectangle. When `Clipping` is `on`, MATLAB does not display any portion of the patch outside the axes rectangle.

`CreateFcn` string

Callback routine executed during object creation. This property defines a callback routine that executes when MATLAB creates a patch object. You must define this property as a default value for patches. For example, the statement,

```
set(0,'DefaultPatchCreateFcn','set(gcf,''DitherMap'',my_dither_
map)')
```

defines a default value on the root level that sets the figure `DitherMap` property whenever you create a patch object. MATLAB executes this routine after setting all properties for the patch created. Setting this property on an existing patch object has no effect.

The handle of the object whose `CreateFcn` is being executed is accessible only through the root `CallbackObject` property, which you can query using `gcbo`.

`DeleteFcn` string

Delete patch callback routine. A callback routine that executes when you delete the patch object (e.g., when you issue a `delete` command or clear the axes (`cla`) or figure (`clf`) containing the patch). MATLAB executes the routine before deleting the object's properties so these values are available to the callback routine.

The handle of the object whose `DeleteFcn` is being executed is accessible only through the root `CallbackObject` property, which you can query using `gcbo`.

DiffuseStrength scalar >= 0 and <= 1

Intensity of diffuse light. This property sets the intensity of the diffuse component of the light falling on the patch. Diffuse light comes from light objects in the axes.

You can also set the intensity of the ambient and specular components of the light on the patch object. See the `AmbientStrength` and `SpecularStrength` properties.

EdgeAlpha `{scalar = 1} | flat | interp`

Transparency of the edges of patch faces. This property can be any of the following:

- `scalar` - A single non-Nan scalar value between `0` and `1` that controls the transparency of all the edges of the object. `1` (the default) is fully opaque and `0` means completely transparent.
- `flat` - The alpha data (`FaceVertexAlphaData`) of each vertex controls the transparency of the edge that follows it.
- `interp` - Linear interpolation of the alpha data (`FaceVertexAlphaData`) at each vertex determines the transparency of the edge.

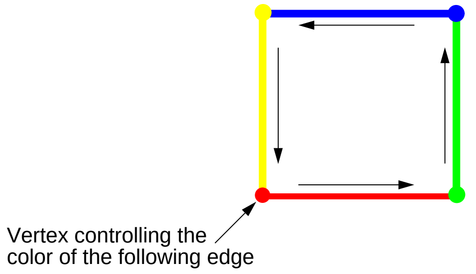
Note that you cannot specify `flat` or `interp` `EdgeAlpha` without first setting `FaceVertexAlphaData` to a matrix containing one alpha value per face (`flat`) or one alpha value per vertex (`interp`).

EdgeColor `{ColorSpec} | none | flat | interp`

Color of the patch edge. This property determines how MATLAB colors the edges of the individual faces that make up the patch.

- `ColorSpec` – A three-element RGB vector or one of MATLAB's predefined names, specifying a single color for edges. The default edge color is black. See `ColorSpec` for more information on specifying color.
- `none` – Edges are not drawn.

- flat – The color of each vertex controls the color of the edge that follows it. This means flat edge coloring is dependent on the order you specify the vertices:



- interp – Linear interpolation of the CData or FaceVertexCData values at the vertices determines the edge color.

EdgeLighting {none} | flat | gouraud | phong

Algorithm used for lighting calculations. This property selects the algorithm used to calculate the effect of light objects on patch edges. Choices are:

- none – Lights do not affect the edges of this object.
- flat – The effect of light objects is uniform across each edge of the patch.
- gouraud – The effect of light objects is calculated at the vertices and then linearly interpolated across the edge lines.
- phong – The effect of light objects is determined by interpolating the vertex normals across each edge line and calculating the reflectance at each pixel. Phong lighting generally produces better results than Gouraud lighting, but takes longer to render.

EraseMode {normal} | none | xor | background

Erase mode. This property controls the technique MATLAB uses to draw and erase patch objects. Alternative erase modes are useful in creating animated sequences, where control of the way individual objects redraw is necessary to improve performance and obtain the desired effect.

- normal – Redraw the affected region of the display, performing the three-dimensional analysis necessary to ensure that all objects are rendered correctly. This mode produces the most accurate picture, but is the slowest.

The other modes are faster, but do not perform a complete redraw and are therefore less accurate.

- `none` – Do not erase the patch when it is moved or destroyed. While the object is still visible on the screen after erasing with `EraseMode` `none`, you cannot print it because MATLAB stores no information about its former location.
- `xor`– Draw and erase the patch by performing an exclusive OR (XOR) with each pixel index of the screen behind it. Erasing the patch does not damage the color of the objects behind it. However, patch color depends on the color of the screen behind it and is correctly colored only when over the axes background `Color`, or the figure background `Color` if the axes `Color` is set to `none`.
- `background` – Erase the patch by drawing it in the axes' background `Color`, or the figure background `Color` if the axes `Color` is set to `none`. This damages objects that are behind the erased patch, but the patch is always properly colored.

Printing with Non-normal Erase Modes. MATLAB always prints figures as if the `EraseMode` of all objects is `normal`. This means graphics objects created with `EraseMode` set to `none`, `xor`, or `background` can look different on screen than on paper. On screen, MATLAB may mathematically combine layers of colors (e.g., XORing a pixel color with that of the pixel behind it) and ignore three-dimensional sorting to obtain greater rendering speed. However, these techniques are not applied to the printed output.

You can use the MATLAB `getframe` command or other screen capture application to create an image of a figure containing non-normal mode objects.

`FaceAlpha` `{scalar = 1} | flat | interp`

Transparency of the patch face. This property can be any of the following:

- A scalar - A single non-NaN scalar value between `0` and `1` that controls the transparency of all the faces of the object. `1` (the default) is fully opaque and `0` is completely transparent (invisible).
- `flat` - The values of the alpha data (`FaceVertexAlphaData`) determine the transparency for each face. The alpha data at the first vertex determines the transparency of the entire face.
- `interp` - Bilinear interpolation of the alpha data (`FaceVertexAlphaData`) at each vertex determine the transparency of each face.

Note that you cannot specify `flat` or `interp` `FaceAlpha` without first setting `FaceVertexAlphaData` to a matrix containing one alpha value per face (`flat`) or one alpha value per vertex (`interp`).

`FaceColor` `{ColorSpec} | none | flat | interp`

Color of the patch face. This property can be any of the following:

- `ColorSpec` – A three-element RGB vector or one of MATLAB's predefined names, specifying a single color for faces. See `ColorSpec` for more information on specifying color.
- `none` – Do not draw faces. Note that edges are drawn independently of faces.
- `flat` – The values of `CData` or `FaceVertexCData` determine the color for each face in the patch. The color data at the first vertex determines the color of the entire face.
- `interp` – Bilinear interpolation of the color at each vertex determines the coloring of each face.

`FaceLighting` `{none} | flat | gouraud | phong`

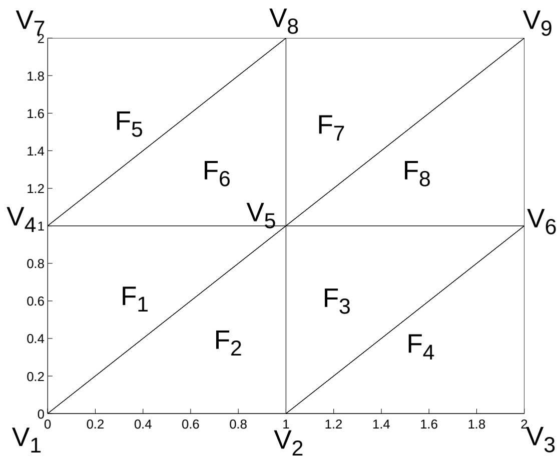
Algorithm used for lighting calculations. This property selects the algorithm used to calculate the effect of light objects on patch faces. Choices are:

- `none` – Lights do not affect the faces of this object.
- `flat` – The effect of light objects is uniform across the faces of the patch. Select this choice to view faceted objects.
- `gouraud` – The effect of light objects is calculated at the vertices and then linearly interpolated across the faces. Select this choice to view curved surfaces.
- `phong` – The effect of light objects is determined by interpolating the vertex normals across each face and calculating the reflectance at each pixel. Select this choice to view curved surfaces. Phong lighting generally produces better results than Gouraud lighting, but takes longer to render.

`Faces` m-by-n matrix

Vertex connection defining each face. This property is the connection matrix specifying which vertices in the `Vertices` property are connected. The `Faces` matrix defines *m* faces with up to *n* vertices each. Each row designates the connections for a single face, and the number of elements in that row that are not `NaN` defines the number of vertices for that face.

The `Faces` and `Vertices` properties provide an alternative way to specify a patch that can be more efficient than using *x*, *y*, and *z* coordinates in most cases. For example, consider the following patch. It is composed of eight triangular faces defined by nine vertices.



`Faces` property

F_1	V_1	V_4	V_5
F_2	V_1	V_5	V_2
F_3	V_2	V_5	V_6
F_4	V_2	V_6	V_3
F_5	V_4	V_7	V_8
F_6	V_4	V_8	V_5
F_7	V_5	V_8	V_9
F8	V_5	V_9	V_6

`Vertices` property

V_1	X_1	Y_1	Z_1
V_2	X_2	Y_2	Z_2
V_3	X_3	Y_3	Z_3
V_4	X_4	Y_4	Z_4
V_5	X_5	Y_5	Z_5
V_6	X_6	Y_6	Z_6
V_7	X_7	Y_7	Z_7
V_8	X_8	Y_8	Z_8
V_9	X_9	Y_9	Z_9

The corresponding `Faces` and `Vertices` properties are shown to the right of the patch. Note how some faces share vertices with other faces. For example, the fifth vertex (V5) is used six times, once each by faces one, two, and three and six, seven, and eight. Without sharing vertices, this same patch requires 24 vertex definitions.

`FaceVertexAlphaData` m-by-1 matrix

Face and vertex transparency data. The `FaceVertexAlphaData` property specifies the tranparency of patches defined by the `Faces` and `Vertices` properties. The interpretation of the values specified for `FaceVertexAlphaData` depends on the dimensions of the data.

`FaceVertexAlphaData` can be one of the following:

- A single value, which applies the same transparency to the entire patch.
- An m-by-1 matrix (where `m` is the number of rows in the `Faces` property), which specifies one transparency value per face.

- An m-by-1 matrix (where m is the number of rows in the `Vertices` property), which specifies one transparency value per vertex.

`FaceVertexCData` matrix

Face and vertex colors. The `FaceVertexCData` property specifies the color of patches defined by the `Faces` and `Vertices` properties, and the values are used when `FaceColor`, `EdgeColor`, `MarkerFaceColor`, or `MarkerEdgeColor` are set appropriately. The interpretation of the values specified for `FaceVertexCData` depends on the dimensions of the data.

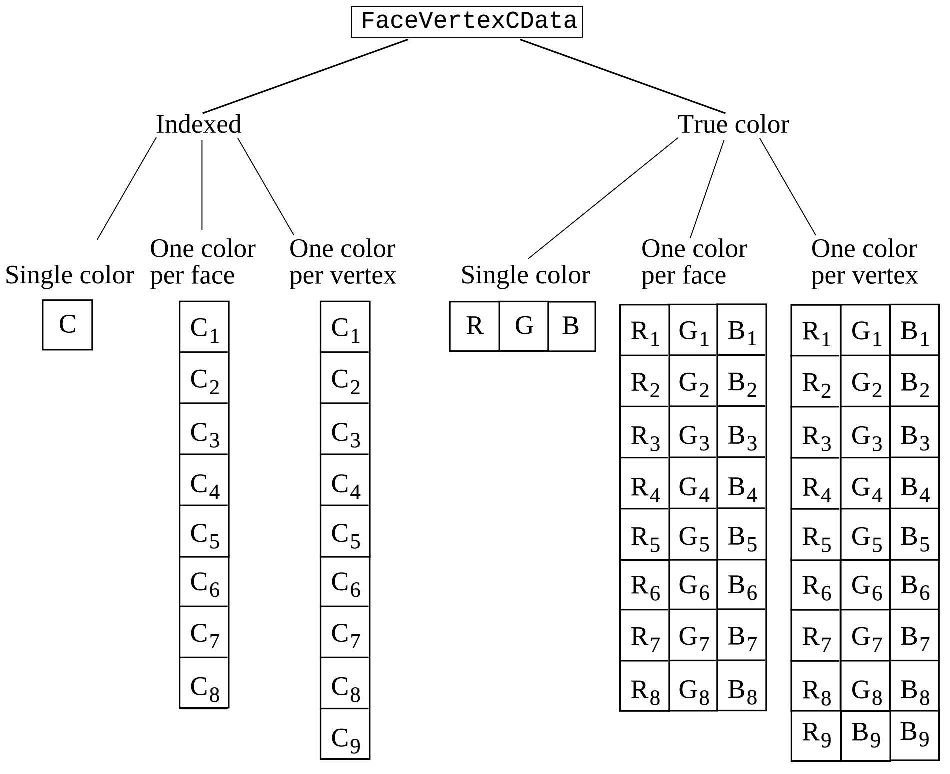
For indexed colors, `FaceVertexCData` can be:

- A single value, which applies a single color to the entire patch
- An *n*-by-1 matrix, where *n* is the number of rows in the `Faces` property, which specifies one color per face
- An *n*-by-1 matrix, where *n* is the number of rows in the `Vertices` property, which specifies one color per vertex

For true colors, `FaceVertexCData` can be:

- A 1-by-3 matrix , which applies a single color to the entire patch
- An *n*-by-3 matrix, where *n* is the number of rows in the `Faces` property, which specifies one color per face
- An *n*-by-3 matrix, where *n* is the number of rows in the `Vertices` property, which specifies one color per vertex

The following diagram illustrates the various forms of the `FaceVertexCData` property for a patch having eight faces and nine vertices. The `CDataMapping` property determines how MATLAB interprets the `FaceVertexCData` property when you specify indexed colors.



HandleVisibility `{on} | callback | off`

Control access to object's handle by command-line users and GUIs. This property determines when an object's handle is visible in its parent's list of children. `HandleVisibility` is useful for preventing command-line users from accidentally drawing into or deleting a figure that contains only user interface devices (such as a dialog box).

Handles are always visible when `HandleVisibility` is `on`.

Setting `HandleVisibility` to `callback` causes handles to be visible from within callback routines or functions invoked by callback routines, but not from within functions invoked from the command line. This provides a means to protect GUIs from command-line users, while allowing callback routines to have complete access to object handles.

Setting `HandleVisibility` to `off` makes handles invisible at all times. This may be necessary when a callback routine invokes a function that might potentially damage the GUI (such as evaluating a user-typed string), and so temporarily hides its own handles during the execution of that function.

When a handle is not visible in its parent's list of children, it cannot be returned by functions that obtain handles by searching the object hierarchy or querying handle properties. This includes `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close`.

When a handle's visibility is restricted using `callback` or `off`, the object's handle does not appear in its parent's `Children` property, figures do not appear in the root's `CurrentFigure` property, objects do not appear in the root's `CallbackObject` property or in the figure's `CurrentObject` property, and axes do not appear in their parent's `Currentaxes` property.

You can set the root `ShowHiddenHandles` property to on to make all handles visible, regardless of their `HandleVisibility` settings (this does not affect the values of the `HandleVisibility` properties).

Handles that are hidden are still valid. If you know an object's handle, you can `set` and `get` its properties, and pass it to any function that operates on handles.

`HitTest` `{on} | off`

Selectable by mouse click. `HitTest` determines if the patch can become the current object (as returned by the `gco` command and the figure `CurrentObject` property) as a result of a mouse click on the patch. If `HitTest` is `off`, clicking on the patch selects the object below it (which maybe the axes containing it).

`Interruptible` `{on} | off`

Callback routine interruption mode. The `Interruptible` property controls whether a patch callback routine can be interrupted by subsequently invoked callback routines. Only callback routines defined for the `ButtonDownFcn` are affected by the `Interruptible` property. MATLAB checks for events that can interrupt a callback routine only when it encounters a `drawnow`, `figure`, `getframe`, or `pause` command in the routine. See the `BusyAction` property for related information.

LineStyle {–} | –– | : | –. | none

Edge linestyle. This property specifies the line style of the patch edges. The following table lists the available line styles.

Symbol	Line Style
–	solid line (default)
––	dashed line
:	dotted line
–.	dash-dot line
none	no line

You can use LineStyle none when you want to place a marker at each point but do not want the points connected with a line (see the Marker property).

LineWidth scalar

Edge line width. The width, in points, of the patch edges (1 point = $^{1}/_{72}$ inch). The default LineWidth is 0.5 points.

Marker character (see table)

Marker symbol. The Marker property specifies marks that locate vertices. You can set values for the Marker property independently from the LineStyle property. The following tables lists the available markers.

Marker Specifier	Description
+	plus sign
o	circle
*	asterisk
.	point
x	cross
s	square

Marker Specifier	Description
d	diamond
^	upward pointing triangle
v	downward pointing triangle
>	right pointing triangle
<	left pointing triangle
p	five-pointed star (pentagram)
h	six-pointed star (hexagram)
none	no marker (default)

MarkerEdgeColor `ColorSpec | none | {auto} | flat`

Marker edge color. The color of the marker or the edge color for filled markers (circle, square, diamond, pentagram, hexagram, and the four triangles). `ColorSpec` defines the color to use. `none` specifies no color, which makes nonfilled markers invisible. `auto` sets `MarkerEdgeColor` to the same color as the `EdgeColor` property.

MarkerFaceColor `ColorSpec | {none} | auto | flat`

Marker face color. The fill color for markers that are closed shapes (circle, square, diamond, pentagram, hexagram, and the four triangles). `ColorSpec` defines the color to use. `none` makes the interior of the marker transparent, allowing the background to show through. `auto` sets the fill color to the axes color, or the figure color, if the axes `Color` property is set to `none`.

MarkerSize size in points

Marker size. A scalar specifying the size of the marker, in points. The default value for `MarkerSize` is six points (1 point = $^1/_{72}$ inch). Note that MATLAB draws the `point` marker at 1/3 of the specified size.

NormalMode `{auto} | manual`

MATLAB-generated or user-specified normal vectors. When this property is `auto`, MATLAB calculates vertex normals based on the coordinate data. If you specify your own vertex normals, MATLAB sets this property to manual and does not generate its own data. See also the `VertexNormals` property.

`Parent` axes handle

Patch's parent. The handle of the patch's parent object. The parent of a patch object is the axes in which it is displayed. You can move a patch object to another axes by setting this property to the handle of the new parent.

`Selected` `on | {off}`

Is object selected? When this property is `on`, MATLAB displays selection handles or a dashed box (depending on the number of faces) if the `SelectionHighlight` property is also `on`. You can, for example, define the `ButtonDownFcn` to set this property, allowing users to select the object with the mouse.

`SelectionHighlight` `{on} | off`

Objects highlight when selected. When the `Selected` property is `on`, MATLAB indicates the selected state by:

- Drawing handles at each vertex for a single-faced patch.
- Drawing a dashed bounding box for a multi-faced patch.

When `SelectionHighlight` is `off`, MATLAB does not draw the handles.

`SpecularColorReflectance` scalar in the range 0 to 1

Color of specularly reflected light. When this property is 0, the color of the specularly reflected light depends on both the color of the object from which it reflects and the color of the light source. When set to 1, the color of the specularly reflected light depends only on the color or the light source (i.e., the light object `Color` property). The proportions vary linearly for values in between.

`SpecularExponent` scalar >= 1

Harshness of specular reflection. This property controls the size of the specular spot. Most materials have exponents in the range of 5 to 20.

`SpecularStrength` scalar >= 0 and <= 1

Intensity of specular light. This property sets the intensity of the specular component of the light falling on the patch. Specular light comes from light objects in the axes.

You can also set the intensity of the ambient and diffuse components of the light on the patch object. See the `AmbientStrength` and `DiffuseStrength` properties.

`Tag` string

User-specified object label. The `Tag` property provides a means to identify graphics objects with a user-specified label. This is particularly useful when constructing interactive graphics programs that would otherwise need to define object handles as global variables or pass them as arguments between callback routines.

For example, suppose you use patch objects to create borders for a group of uicontrol objects and want to change the color of the borders in a uicontrol's callback routine. You can specify a `Tag` with the patch definition:

```
patch(X,Y,'k','Tag','PatchBorder')
```

Then use `findobj` in the uicontrol's callback routine to obtain the handle of the patch and set its `FaceColor` property.

```
set(findobj('Tag','PatchBorder'),'FaceColor','w')
```

`Type` string (read only)

Class of the graphics object. For patch objects, `Type` is always the string `'patch'`.

`UIContextMenu` handle of a uicontextmenu object

Associate a context menu with the patch. Assign this property the handle of a uicontextmenu object created in the same figure as the patch. Use the `uicontextmenu` function to create the context menu. MATLAB displays the context menu whenever you right-click over the patch.

`UserData` matrix

User-specified data. Any matrix you want to associate with the patch object. MATLAB does not use this data, but you can access it using `set` and `get`.

`VertexNormals` matrix

Surface normal vectors. This property contains the vertex normals for the patch. MATLAB generates this data to perform lighting calculations. You can supply your own vertex normal data, even if it does not match the coordinate data. This can be useful to produce interesting lighting effects.

Vertices matrix

Vertex coordinates. A matrix containing the *x*-, *y*-, *z*-coordinates for each vertex. See the `Faces` property for more information.

Visible `{on} | off`

Patch object visibility. By default, all patches are visible. When set to `off`, the patch is not visible, but still exists and you can query and set its properties.

XData vector or matrix

X-coordinates. The *x*-coordinates of the points at the vertices of the patch. If `XData` is a matrix, each column represents the *x*-coordinates of a single face of the patch. In this case, `XData`, `YData`, and `ZData` must have the same dimensions.

YData vector or matrix

Y-coordinates. The *y*-coordinates of the points at the vertices of the patch. If `YData` is a matrix, each column represents the *y*-coordinates of a single face of the patch. In this case, `XData`, `YData`, and `ZData` must have the same dimensions.

ZData vector or matrix

Z-coordinates. The *z*-coordinates of the points at the vertices of the patch. If `ZData` is a matrix, each column represents the *z*-coordinates of a single face of the patch. In this case, `XData`, `YData`, and `ZData` must have the same dimensions.

See Also

`patch`

path

Purpose View or change the MATLAB directory search path

Graphical Interface As an alternative to the `path` function, use the **Set Path** dialog box. To open it, select **Set Path** from the **File** menu in the MATLAB desktop.

Syntax

```
path
path newpath
path(path,'newpath')
path('newpath',path)
p = path(...)
```

Description `path` displays the current MATLAB search path. The initial search path list is defined by `toolbox/local/pathdef.m`.

`path newpath` changes the search path to be comprised of those directories named in the string, `'newpath'`.

`path(path,'newpath')` appends a new directory to the current search path.

`path('newpath',path)` prepends a new directory to the current search path.

`p = path(...)` returns the specified path in string variable `p`.

Remarks For more information on how MATLAB uses the directory search path, see How Functions Work and How MATLAB Determines Which Method to Call.

Note Save any M-files you create and any MATLAB-supplied M-files that you edit in a directory that is not in the `MATLAB` directory tree. If you keep your files in the `MATLAB` directory tree, they may be overwritten when you install a new version of MATLAB. Also note that locations of files in the `MATLAB/toolbox` directory tree are loaded and cached in memory at the beginning of each MATLAB session to improve performance. If you do save a new or edited file in the `MATLAB/toolbox` directory tree, restart MATLAB or use the `rehash` function to reload the directory and update the cache before you use the file.

Examples

To add a new directory to the search path on Windows,

```
path(path,'c:tools\goodstuff')
```

To add a new directory to the search path on UNIX,

```
path(path,'/home/tools/goodstuff')
```

See Also

`addpath`, `genpath`, `cd`, `dir`, `partialpath`, `rehash`, `rmpath`, `what`

pathtool

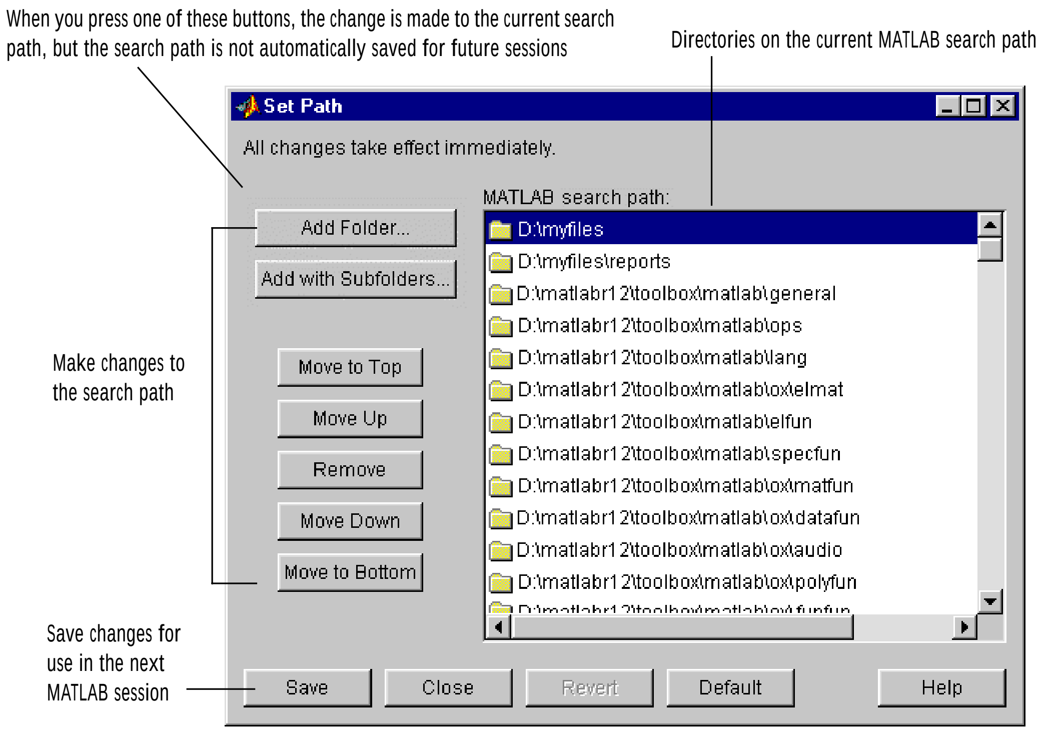
Purpose Open **Set Path** dialog box to view and change MATLAB path

Graphical Interface As an alternative to the `pathtool` function, select **Set Path** from the **File** menu in the MATLAB desktop.

Syntax

```
pathtool
```

Description `pathtool` opens the Set Path dialog box, a graphical interface you use to view and modify the MATLAB search path, as well as see files on the path.



See Also `addpath`, `edit`, `path`, `rmpath`, `workspace`

"Setting the Search Path"

Purpose Halt execution temporarily

Syntax

```
pause
pause(n)
pause on
pause off
```

Description `pause`, by itself, causes M-files to stop and wait for you to press any key before continuing.

`pause(n)` pauses execution for `n` seconds before continuing, where `n` can be any real number. The resolution of the clock is platform specific. A fractional pause of 0.01 seconds should be supported on most platforms.

`pause on` allows subsequent pause commands to pause execution.

`pause off` ensures that any subsequent `pause` or `pause(n)` statements do not pause execution. This allows normally interactive scripts to run unattended.

See Also `drawnow`

pbaspect

Purpose Set or query the plot box aspect ratio

Syntax

```
pbaspect
pbaspect([aspect_ratio])
pbaspect('mode')
pbaspect('auto')
pbaspect('manual')
pbaspect(axes_handle,...)
```

Description The plot box aspect ratio determines the relative size of the x-, y-, and z-axes.

`pbaspect` with no arguments returns the plot box aspect ratio of the current axes.

`pbaspect([aspect_ratio])` sets the plot box aspect ratio in the current axes to the specified value. Specify the aspect ratio as three relative values representing the ratio of the x-, y-, and z-axes size. For example, a value of `[1 1 1]` (the default) means the plot box is a cube (although with stretch-to-fill enabled, it may not appear as a cube). See Remarks.

`pbaspect('mode')` returns the current value of the plot box aspect ratio mode, which can be either `auto` (the default) or `manual`. See Remarks.

`pbaspect('auto')` sets the plot box aspect ratio mode to `auto`.

`pbaspect('manual')` sets the plot box aspect ratio mode to `manual`.

`pbaspect(axes_handle,...)` performs the set or query on the axes identified by the first argument, `axes_handle`. If you do not specify an axes handle, `pbaspect` operates on the current axes.

Remarks `pbaspect` sets or queries values of the axes object `PlotBoxAspectRatio` and `PlotBoxAspectRatioMode` properties.

When the plot box aspect ratio mode is `auto`, MATLAB sets the ratio to `[1 1 1]`, but may change it to accommodate manual settings of the data aspect ratio, camera view angle, or axis limits. See the axes `DataAspectRatio` property for a table listing the interactions between various properties.

Setting a value for the plot box aspect ratio or setting the plot box aspect ratio mode to `manual` disables MATLAB's stretch-to-fill feature (stretching of the axes to fit the window). This means setting the plot box aspect ratio to its current value,

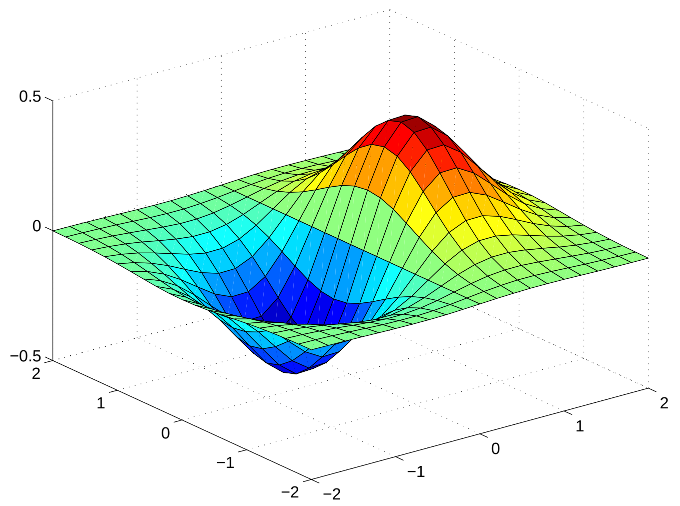
```
pbaspect(pbaspect)
```

can cause a change it the way the graphs look. See the Remarks section of the axes reference description and the "Aspect Ratio" section in the *Using MATLAB Graphics* manual for a discussion of stretch-to-fill.

Examples

The following surface plot of the function $z = xe^{(-x^2 - y^2)}$ is useful to illustrate the plot box aspect ratio. First plot the function over the range $-2 \le x \le 2$, $-2 \le y \le 2$,

```
[x,y] = meshgrid([-2:.2:2]);
z = x.*exp(-x.^2 - y.^2);
surf(x,y,z)
```



Querying the plot box aspect ratio shows that the plot box is square.

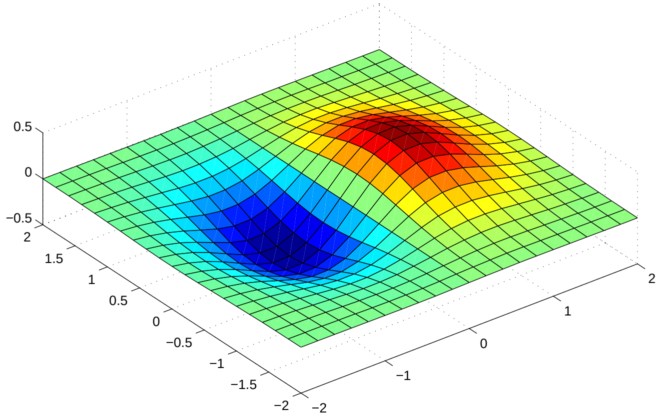
```
pbaspect
ans =
     1   1   1
```

It is also interesting to look at the data aspect ratio selected by MATLAB.

```
daspect
ans =
     4   4   1
```

To illustrate the interaction between the plot box and data aspect ratios, set the data aspect ratio to [1 1 1] and again query the plot box aspect ratio.

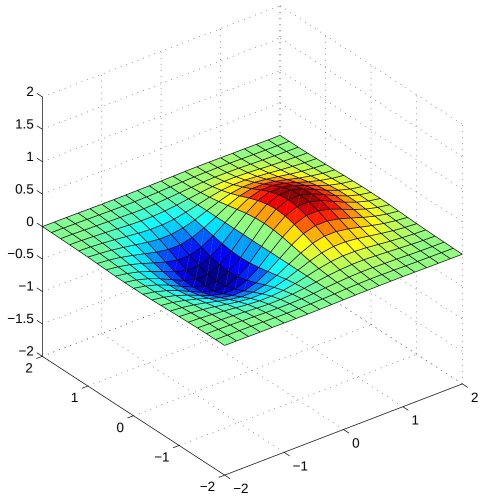
```
daspect([1 1 1])
```




```
pbaspect
ans =
     4   4   1
```

The plot box aspect ratio has changed to accommodate the specified data aspect ratio. Now suppose you want the plot box aspect ratio to be [1 1 1] as well.

```
pbaspect([1 1 1])
```

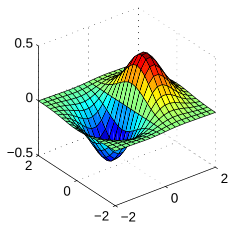


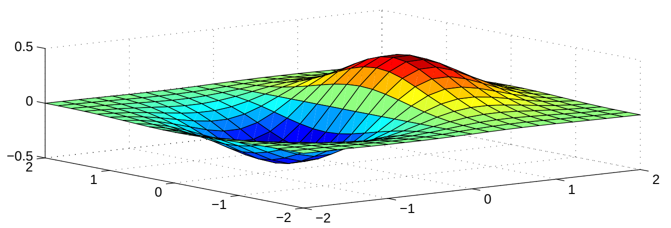
Notice how MATLAB changed the axes limits because of the constraints introduced by specifying both the plot box and data aspect ratios.

You can also use `pbaspect` to disable stretch-to-fill. For example, displaying two subplots in one figure can give surface plots a squashed appearance. Disabling stretch-to-fill.

```
upper_plot = subplot(211);
surf(x,y,z)
lower_plot = subplot(212);
surf(x,y,z)
pbaspect(upper_plot,'manual')
```

pbaspect





See Also axis, daspect, xlim, ylim, zlim

The axes properties DataAspectRatio, PlotBoxAspectRatio, XLim, YLim, ZLim

The “Aspect Ratio” section in the *Using MATLAB Graphics* manual.

Purpose Preconditioned Conjugate Gradients method

Syntax

```
x = pcg(A,b)
pcg(A,b,tol)
pcg(A,b,tol,maxit)
pcg(A,b,tol,maxit,M)
pcg(A,b,tol,maxit,M1,M2)
pcg(A,b,tol,maxit,M1,M2,x0)
pcg(A,b,tol,maxit,M1,M2,x0,p1,p2,...)
[x,flag] = pcg(A,b,tol,maxit,M1,M2,x0,p1,p2,...)
[x,flag,relres] = pcg(A,b,tol,maxit,M1,M2,x0,p1,p2,...)
[x,flag,relres,iter] = pcg(A,b,tol,maxit,M1,M2,x0,p1,p2,...)
[x,flag,relres,iter,resvec] =
   pcg(A,b,tol,maxit,M1,M2,x0,p1,p2,...)
```

Description

`x = pcg(A,b)` attempts to solve the system of linear equations `A*x=b` for `x`. The `n`-by-`n` coefficient matrix `A` must be symmetric and positive definite and the column vector `b` must have length `n`. `A` can be a function `afun` such that `afun(x)` returns `A*x`.

If `pcg` converges, a message to that effect is displayed. If `pcg` fails to converge after the maximum number of iterations or halts for any reason, a warning message is printed displaying the relative residual `norm(b-A*x)/norm(b)` and the iteration number at which the method stopped or failed.

`pcg(A,b,tol)` specifies the tolerance of the method. If `tol` is `[]`, then `pcg` uses the default, `1e-6`.

`pcg(A,b,tol,maxit)` specifies the maximum number of iterations. If `maxit` is `[]`, then `pcg` uses the default, `min(n,20)`.

`pcg(A,b,tol,maxit,M)` and `pcg(A,b,tol,maxit,M1,M2)` use symmetric positive definite preconditioner `M` or `M = M1*M2` and effectively solve the system `inv(M)*A*x = inv(M)*b` for `x`. If `M` is `[]` then `pcg` applies no preconditioner. `M` can be a function that returns `M\x`.

`pcg(A,b,tol,maxit,M1,M2,x0)` specifies the initial guess. If `x0` is `[]`, then `pcg` uses the default, an all-zero vector.

pcg(afun,b,tol,maxit,m1fun,m2fun,x0,p1,p2,...) passes parameters p1,p2,... to functions afun(x,p1,p2,...), m1fun(x,p1,p2,...), and m2fun(x,p1,p2,...).

[x,flag] = pcg(A,b,tol,maxit,M1,M2,x0) also returns a convergence flag.

Flag	Convergence
0	pcg converged to the desired tolerance tol within maxit iterations.
1	pcg iterated maxit times but did not converge.
2	Preconditioner M was ill-conditioned.
3	pcg stagnated. (Two consecutive iterates were the same.)
4	One of the scalar quantities calculated during pcg became too small or too large to continue computing.

Whenever flag is not 0, the solution x returned is that with minimal norm residual computed over all the iterations. No messages are displayed if the flag output is specified.

[x,flag,relres] = pcg(A,b,tol,maxit,M1,M2,x0) also returns the relative residual norm(b-A*x)/norm(b). If flag is 0, relres <= tol.

[x,flag,relres,iter] = pcg(A,b,tol,maxit,M1,M2,x0) also returns the iteration number at which x was computed, where 0 <= iter <= maxit.

[x,flag,relres,iter,resvec] = pcg(A,b,tol,maxit,M1,M2,x0) also returns a vector of the residual norms at each iteration including norm(b-A*x0).

Examples

Example 1.

```
A = gallery('wilk',21);
b = sum(A,2);
tol = 1e-12;
maxit = 15;
M = diag([10:-1:1 1 1:10]);
```

```
[x,flag,rr,iter,rv] = pcg(A,b,tol,maxit,M);
```

Alternatively, use this one-line matrix-vector product function

```
function y = afun(x,n)
y = [0;
     x(1:n-1)] + [((n-1)/2:-1:0)';
     (1:(n-1)/2)'].*x + [x(2:n);
     0];
```

and this one-line preconditioner backsolve function

```
function y = mfun(r,n)
y = r ./ [((n-1)/2:-1:1)'; 1; (1:(n-1)/2)'];
```

as inputs to `pcg`

```
[x1,flag1,rr1,iter1,rv1] = pcg(@afun,b,tol,maxit,@mfun,...
                              [],[],21);
```

Example 2.

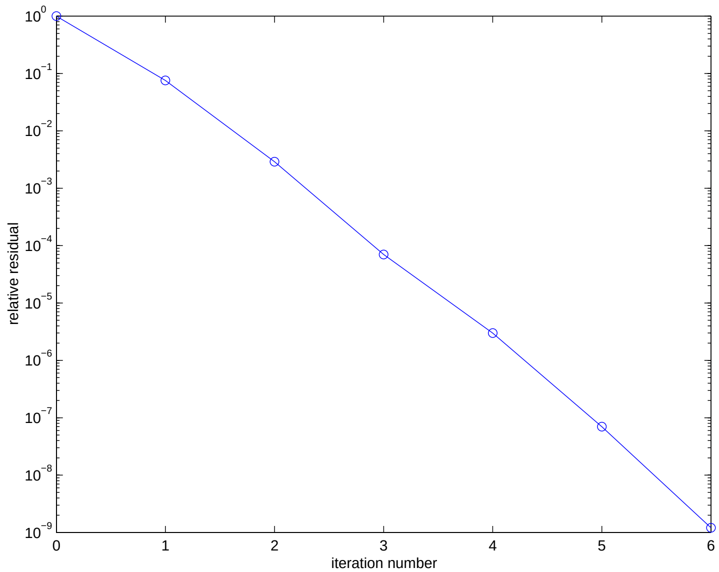
```
A = delsq(numgrid('C',25));
b = ones(length(A),1);
[x,flag] = pcg(A,b)
```

`flag` is `1` because `pcg` does not converge to the default tolerance of `1e-6` within the default 20 iterations.

```
R = cholinc(A,1e-3);
[x2,flag2,relres2,iter2,resvec2] = pcg(A,b,1e-8,10,R',R)
```

`flag2` is `0` because `pcg` converges to the tolerance of `1.2e-9` (the value of `relres2`) at the sixth iteration (the value of `iter2`) when preconditioned by the incomplete Cholesky factorization with a drop tolerance of `1e-3`. `resvec2(1) = norm(b)` and `resvec2(7) = norm(b-A*x2)`. You can follow the progress of `pcg` by plotting the relative residuals at each iteration starting from the initial estimate (iterate number 0).

```
semilogy(0:iter2,resvec2/norm(b),'-o')
xlabel('iteration number')
ylabel('relative residual')
```



See Also

`bicg`, `bicgstab`, `cgs`, `cholinc`, `gmres`, `lsqr`, `minres`, `qmr`, `symmlq`

`@` (function handle), `\` (backslash)

References

[1] Barrett, R., M. Berry, T. F. Chan, et al., *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*, SIAM, Philadelphia, 1994.

Purpose

Piecewise Cubic Hermite Interpolating Polynomial (PCHIP)

Syntax

```
yi = pchip(x,y,xi)
pp = pchip(x,y)
```

Description

`yi = pchip(x,y,xi)` returns vector `yi` containing elements corresponding to the elements of `xi` and determined by piecewise cubic interpolation within vectors `x` and `y`. The vector `x` specifies the points at which the data `y` is given. If `y` is a matrix, then the interpolation is performed for each column of `y` and `yi` is `length(xi)`-by-`size(y,2)`.

`pp = pchip(x,y)` returns a piecewise polynomial structure for use by `ppval`. `x` can be a row or column vector. `y` is a row or column vector of the same length as `x`, or a matrix with `length(x)` columns.

`pchip` finds values of an underlying interpolating function $P(x)$ at intermediate points, such that:

- On each subinterval $x_k \le x \le x_{k+1}$, $P(x)$ is the cubic Hermite interpolant to the given values and certain slopes at the two endpoints.
- $P(x)$ interpolates y, i.e., $P(x_j) = y_j$, and the first derivative $P'(x)$ is continuous. $P''(x)$ is probably not continuous; there may be jumps at the x_j.
- The slopes at the x_j are chosen in such a way that $P(x)$ preserves the shape of the data and respects monotonicity. This means that, on intervals where the data are monotonic, so is $P(x)$; at points where the data has a local extremum, so does $P(x)$.

Note If y is a matrix, $P(x)$ satisfies the above for each column of y.

Remarks

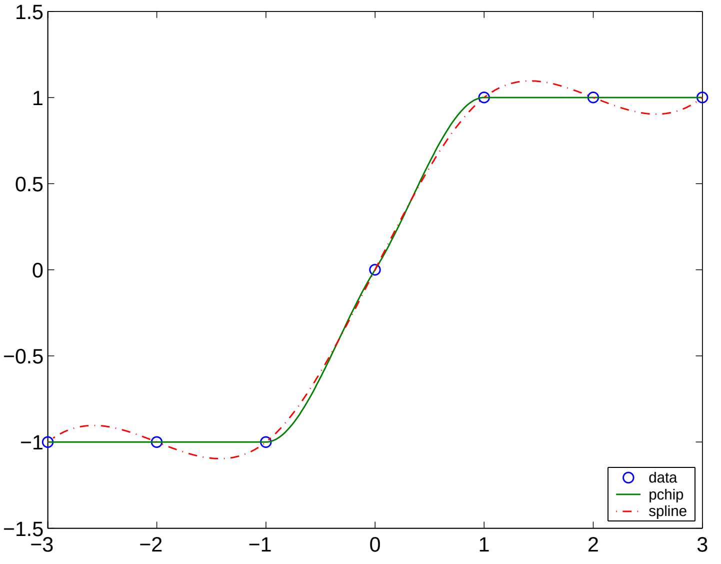
`spline` constructs $S(x)$ in almost the same way `pchip` constructs $P(x)$. However, `spline` chooses the slopes at the x_j differently, namely to make even $S''(x)$ continuous. This has the following effects:

- `spline` produces a smoother result, i.e. $S''(x)$ is continuous.
- `spline` produces a more accurate result if the data consists of values of a smooth function.

- pchip has no overshoots and less oscillation if the data are not smooth.
- pchip is less expensive to set up.
- The two are equally expensive to evaluate.

Examples

```
x = -3:3;
y = [-1 -1 -1 0 1 1 1];
t = -3:.01:3;
p = pchip(x,y,t);
s = spline(x,y,t);
plot(x,y,'o',t,p,'-',t,s,'-.')
legend('data','pchip','spline',4)
```



See Also

interp1, spline, ppval

References

[1] Fritsch, F. N. and R. E. Carlson, "Monotone Piecewise Cubic Interpolation," *SIAM J. Numerical Analysis*, Vol. 17, 1980, pp.238-246.

[2] Kahaner, David, Cleve Moler, Stephen Nash, *Numerical Methods and Software*, Prentice Hall, 1988.

Purpose

Create preparsed pseudocode file (P-file)

Syntax

```
pcode fun
pcode *.m
pcode fun1 fun2 ...
pcode... -inplace
```

Description

`pcode` *`fun`* parses the M-file `fun.m` into the P-file `fun.p` and puts it into the current directory. The original M-file can be anywhere on the search path.

`pcode *.m` creates P-files for all the M-files in the current directory.

`pcode` *`fun1 fun2`* `...` creates P-files for the listed functions.

`pcode... -inplace` creates P-files in the same directory as the M-files. An error occurs if the files can’t be created.

pcolor

Purpose

Pseudocolor plot

Syntax

```
pcolor(C)
pcolor(X,Y,C)
h = pcolor(...)
```

Description

A pseudocolor plot is a rectangular array of cells with colors determined by `C`. MATLAB creates a pseudocolor plot by using each set of four adjacent points in `C` to define a surface patch (i.e., cell).

`pcolor(C)` draws a pseudocolor plot. The elements of `C` are linearly mapped to an index into the current colormap. The mapping from `C` to the current colormap is defined by `colormap` and `caxis`.

`pcolor(X,Y,C)` draws a pseudocolor plot of the elements of `C` at the locations specified by `X` and `Y`. The plot is a logically rectangular, two-dimensional grid with vertices at the points `[X(i,j), Y(i,j)]`. `X` and `Y` are vectors or matrices that specify the spacing of the grid lines. If `X` and `Y` are vectors, `X` corresponds to the columns of `C` and `Y` corresponds to the rows. If `X` and `Y` are matrices, they must be the same size as `C`.

`h = pcolor(...)` returns a handle to a surface graphics object.

Remarks

A pseudocolor plot is a flat surface plot viewed from above. `pcolor(X,Y,C)` is the same as viewing `surf(X,Y,0*Z,C)` using `view([0 90])`.

When you use `shading faceted` or `shading flat`, the constant color of each cell is the color associated with the corner having the smallest *x*-*y* coordinates. Therefore, `C(i,j)` determines the color of the cell in the *i*th row and *j*th column. The last row and column of `C` are not used.

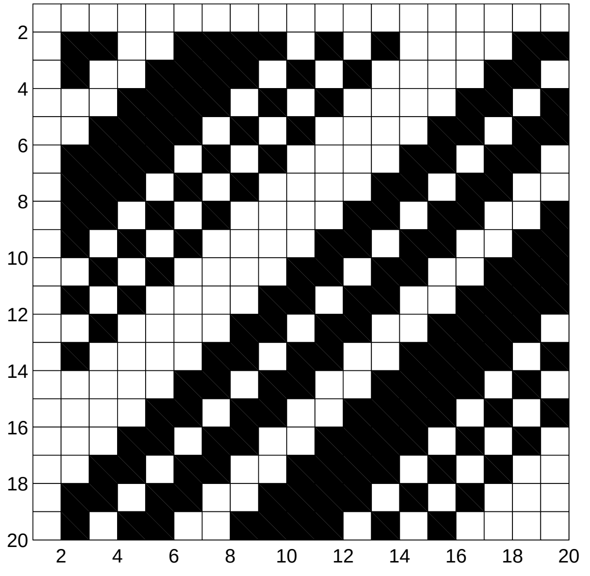
When you use `shading interp`, each cell's color results from a bilinear interpolation of the colors at its four vertices and all elements of `C` are used.

Examples

A Hadamard matrix has elements that are `+1` and `–1`. A colormap with only two entries is appropriate when displaying a pseudocolor plot of this matrix.

```
pcolor(hadamard(20))
colormap(gray(2))
axis ij
```

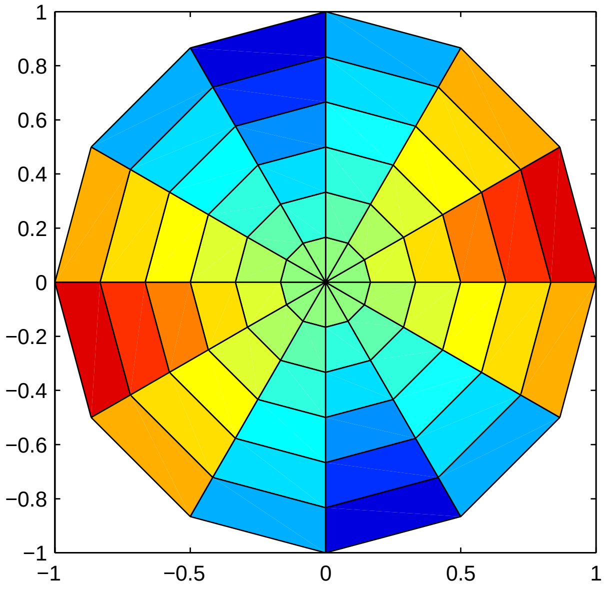
```
axis square
```



A simple color wheel illustrates a polar coordinate system.

```
n = 6;
r = (0:n)'/n;
theta = pi*(-n:n)/n;
X = r*cos(theta);
Y = r*sin(theta);
C = r*cos(2*theta);
pcolor(X,Y,C)
```

```
axis equal tight
```



Algorithm

The number of vertex colors for `pcolor(C)` is the same as the number of cells for `image(C)`. `pcolor` differs from `image` in that `pcolor(C)` specifies the colors of vertices, which are scaled to fit the colormap; changing the axes `clim` property changes this color mapping. `image(C)` specifies the colors of cells and directly indexes into the colormap without scaling. Additionally, `pcolor(X,Y,C)` can produce parametric grids, which is not possible with `image`.

See Also

`caxis`, `image`, `mesh`, `shading`, `surf`, `view`

Purpose

Solve initial-boundary value problems for systems of parabolic and elliptic partial differential equations (PDEs) in one space variable and time

Syntax

```
sol = pdepe(m,pdefun,icfun,bcfun,xmesh,tspan)
sol = pdepe(m,pdefun,icfun,bcfun,xmesh,tspan,options)
sol = pdepe(m,pdefun,icfun,bcfun,xmesh,tspan,options,p1,p2...)
```

Arguments

`m`	A parameter corresponding to the symmetry of the problem. `m` can be slab = `0`, cylindrical = `1`, or spherical = `2`.
`pdefun`	A function that defines the components of the PDE.
`icfun`	A function that defines the initial conditions.
`bcfun`	A function that defines the boundary conditions.
`xmesh`	A vector [`x0`, `x1`, ..., `xn`] specifying the points at which a numerical solution is requested for every value in `tspan`. The elements of `xmesh` must satisfy `x0` < `x1` < ... < `xn`. The length of `xmesh` must be >= `3`.
`tspan`	A vector [`t0`, `t1`, ..., `tf`] specifying the points at which a solution is requested for every value in `xmesh`. The elements of `tspan` must satisfy `t0` < `t1` < ... < `tf`. The length of `tspan` must be >= `3`.
`options`	Some options of the underlying ODE solver are available in `pdepe`: `RelTol`, `AbsTol`, `NormControl`, `InitialStep`, and `MaxStep`. In most cases, default values for these options provide satisfactory solutions. See `odeset` for details.
`p1,p2,...`	Optional parameters to be passed to `pdefun`, `icfun`, and `bcfun`.

Description

`sol = pdepe(m,pdefun,icfun,bcfun,xmesh,tspan)` solves initial-boundary value problems for systems of parabolic and elliptic PDEs in the one space variable x and time t. The ordinary differential equations (ODEs) resulting from discretization in space are integrated to obtain approximate solutions at times specified in `tspan`. The `pdepe` function returns values of the solution on a mesh provided in `xmesh`.

pdepe solves PDEs of the form:

$$c\left(x, t, u, \frac{\partial u}{\partial x}\right)\frac{\partial u}{\partial t} = x^{-m}\frac{\partial}{\partial x}\left(x^m f\left(x, t, u, \frac{\partial u}{\partial x}\right)\right) + s\left(x, t, u, \frac{\partial u}{\partial x}\right) \tag{2-1}$$

The PDEs hold for $t_0 \le t \le t_f$ and $a \le x \le b$. The interval $[a, b]$ must be finite. m can be 0, 1, or 2, corresponding to slab, cylindrical, or spherical symmetry, respectively. If $m > 0$, then a must be >= 0.

In Equation 2-1, $f(x, t, u, \partial u/\partial x)$ is a flux term and $s(x, t, u, \partial u/\partial x)$ is a source term. The coupling of the partial derivatives with respect to time is restricted to multiplication by a diagonal matrix $c(x, t, u, \partial u/\partial x)$. The diagonal elements of this matrix are either identically zero or positive. An element that is identically zero corresponds to an elliptic equation and otherwise to a parabolic equation. There must be at least one parabolic equation. An element of c that corresponds to a parabolic equation can vanish at isolated values of x if those values of x are mesh points. Discontinuities in c and/or s due to material interfaces are permitted provided that a mesh point is placed at each interface.

For $t = t_0$ and all x, the solution components satisfy initial conditions of the form

$$u(x, t_0) = u_0(x) \tag{2-2}$$

For all t and either $x = a$ or $x = b$, the solution components satisfy a boundary condition of the form

$$p(x, t, u) + q(x, t)\, f\left(x, t, u, \frac{\partial u}{\partial x}\right) = 0 \tag{2-3}$$

Elements of q are either identically zero or never zero. Note that the boundary conditions are expressed in terms of the flux f rather than $\partial u/\partial x$. Also, of the two coefficients, only p can depend on u.

In the call `sol = pdepe(m,pdefun,icfun,bcfun,xmesh,tspan)`:

- `m` corresponds to m.
- `xmesh(1)` and `xmesh(end)` correspond to a and b.
- `tspan(1)` and `tspan(end)` correspond to t_0 and t_f.

- `pdefun` computes the terms c, f, and s (Equation 2-1). It has the form

  ```
  [c,f,s] = pdefun(x,t,u,dudx)
  ```

  The input arguments are scalars `x` and `t` and vectors `u` and `dudx` that approximate the solution u and its partial derivative with respect to x, respectively. `c`, `f`, and `s` are column vectors. `c` stores the diagonal elements of the matrix c (Equation 2-1).
- `icfun` evaluates the initial conditions. It has the form

  ```
  u = icfun(x)
  ```

  When called with an argument `x`, `icfun` evaluates and returns the initial values of the solution components at `x` in the column vector `u`.
- `bcfun` evaluates the terms p and q of the boundary conditions (Equation 2-3). It has the form

  ```
  [pl,ql,pr,qr] = bcfun(xl,ul,xr,ur,t)
  ```

  `ul` is the approximate solution at the left boundary `xl` = a and `ur` is the approximate solution at the right boundary `xr` = b. `pl` and `ql` are column vectors corresponding to p and q evaluated at `xl`, similarly `pr` and `qr` correspond to `xr`. When $m > 0$ and $a = 0$, boundedness of the solution near $x = 0$ requires that the flux f vanish at $a = 0$. `pdepe` imposes this boundary condition automatically and it ignores values returned in `pl` and `ql`.

`pdepe` returns the solution as a multidimensional array `sol`. u_i = `ui` = `sol(:,:,i)` is an approximation to the `i`th component of the solution vector u. The element `ui(j,k)` = `sol(j,k,i)` approximates u_i at (t, x) = (`tspan(j)`,`xmesh(k)`).

`ui` = `sol(j,:,i)` approximates component `i` of the solution at time `tspan(j)` and mesh points `xmesh(:)`. Use `pdeval` to compute the approximation and its partial derivative $\partial u_i / \partial x$ at points not included in `xmesh`. See `pdeval` for details.

`sol = pdepe(m,pdefun,icfun,bcfun,xmesh,tspan,options)` solves as above with default integration parameters replaced by values in `options`, an argument created with the `odeset` function. Only some of the options of the underlying ODE solver are available in `pdepe`: `RelTol`, `AbsTol`, `NormControl`, `InitialStep`, and `MaxStep`. The defaults obtained by leaving off the input argument `options` will generally be satisfactory. See `odeset` for details.

`sol = pdepe(m,pdefun,icfun,bcfun,xmesh,tspan,options,p1,p2...)` passes the additional parameters `p1`, `p2`, ... to the functions `pdefun`, `icfun`, and `bcfun`. Use `options = []` as a placeholder if no options are set.

Remarks

- The arrays `xmesh` and `tspan` play different roles in `pdepe`.

  `tspan` – The `pdepe` function performs the time integration with an ODE solver that selects both the time step and formula dynamically. The elements of `tspan` merely specify where you want answers and the cost depends weakly on the length of `tspan`.

  `xmesh` – Second order approximations to the solution are made on the mesh specified in `xmesh`. Generally, it is best to use closely spaced mesh points where the solution changes rapidly. `pdepe` does *not* select the mesh in x automatically. You must provide an appropriate fixed mesh in `xmesh`. The cost depends strongly on the length of `xmesh`. When $m > 0$, it is not necessary to use a fine mesh near $x = 0$ to account for the coordinate singularity.

- The time integration is done with `ode15s`. `pdepe` exploits the capabilities of `ode15s` for solving the differential-algebraic equations that arise when Equation 2-1 contains elliptic equations, and for handling Jacobians with a specified sparsity pattern.

- After discretization, elliptic equations give rise to algebraic equations. If the elements of the initial conditions vector that correspond to elliptic equations are not "consistent" with the discretization, `pdepe` tries to adjust them before beginning the time integration. For this reason, the solution returned for the initial time may have a discretization error comparable to that at any other time. If the mesh is sufficiently fine, `pdepe` can find consistent initial conditions close to the given ones. If `pdepe` displays a message that it has difficulty finding consistent initial conditions, try refining the mesh.

  No adjustment is necessary for elements of the initial conditions vector that correspond to parabolic equations.

Examples

Example 1. This example illustrates the straightforward formulation, computation, and plotting of the solution of a single PDE.

$$\pi^2 \frac{\partial u}{\partial t} = \frac{\partial}{\partial x}\left(\frac{\partial u}{\partial x}\right)$$

This equation holds on an interval $0 \le x \le 1$ for times $t \ge 0$.

The PDE satisfies the initial condition

$$u(x, 0) = \sin \pi x$$

and boundary conditions

$$u(0, t) \equiv 0$$

$$\pi e^{-t} + \frac{\partial u}{\partial x}(1, t) = 0$$

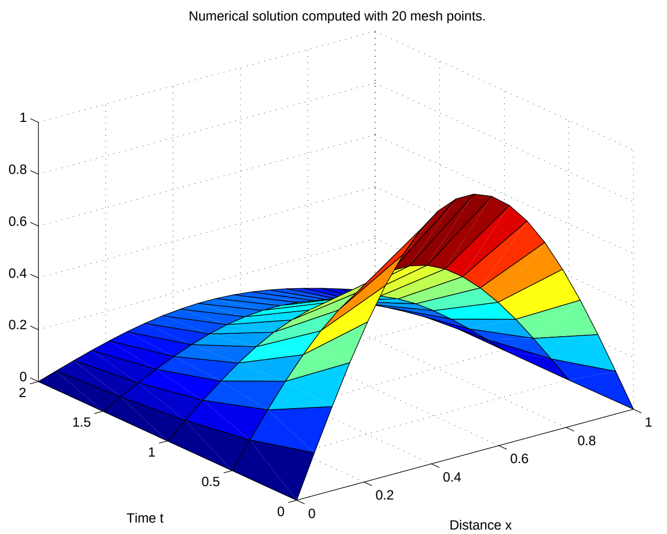
It is convenient to use subfunctions to place all the functions required by pdepe in a single M-file.

```
function pdex1

m = 0;
x = linspace(0,1,20);
t = linspace(0,2,5);

sol = pdepe(m,@pdex1pde,@pdex1ic,@pdex1bc,x,t);
% Extract the first solution component as u.
u = sol(:,:,1);

% A surface plot is often a good way to study a solution.
surf(x,t,u)
title('Numerical solution computed with 20 mesh points.')
xlabel('Distance x')
ylabel('Time t')

% A solution profile can also be illuminating.
figure
plot(x,u(end,:))
title('Solution at t = 2')
xlabel('Distance x')
```

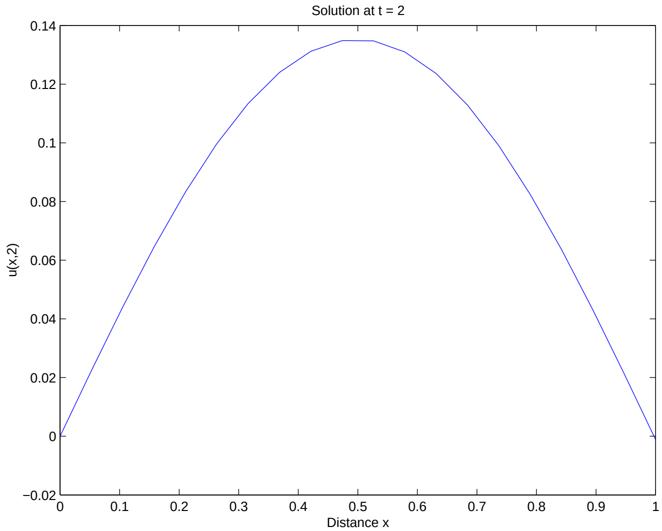
```
ylabel('u(x,2)')
% --------------------------------------------------------------
function [c,f,s] = pdex1pde(x,t,u,DuDx)
c = pi^2;
f = DuDx;
s = 0;
% --------------------------------------------------------------
function u0 = pdex1ic(x)
u0 = sin(pi*x);
% --------------------------------------------------------------
function [pl,ql,pr,qr] = pdex1bc(xl,ul,xr,ur,t)
pl = ul;
ql = 0;
pr = pi * exp(-t);
qr = 1;
```

In this example, the PDE, initial condition, and boundary conditions are coded in subfunctions `pdex1pde`, `pdex1ic`, and `pdex1bc`.

The surface plot shows the behavior of the solution.



The following plot shows the solution profile at the final value of `t` (i.e., `t = 2`).



Example 2. This example illustrates the solution of a system of PDEs. The problem has boundary layers at both ends of the interval. The solution changes rapidly for small t.

The PDEs are

$$\frac{\partial u_1}{\partial t} = 0.024\frac{\partial^2 u_1}{\partial x^2} - F(u_1 - u_2)$$

$$\frac{\partial u_2}{\partial t} = 0.170\frac{\partial^2 u_2}{\partial x^2} + F(u_1 - u_2)$$

where $F(y) = \exp(5.73y) - \exp(-11.46y)$.

This equation holds on an interval $0 \le x \le 1$ for times $t \ge 0$.

The PDE satisfies the initial conditions

$$u_1(x, 0) \equiv 1$$
$$u_2(x, 0) \equiv 0$$

and boundary conditions

$$\frac{\partial u_1}{\partial x}(0, t) \equiv 0$$
$$u_2(0, t) \equiv 0$$
$$u_1(1, t) \equiv 1$$
$$\frac{\partial u_2}{\partial x}(1, t) \equiv 0$$

In the form expected by pdepe, the equations are

$$\begin{bmatrix} 1 \\ 1 \end{bmatrix} .* \frac{\partial}{\partial t}\begin{bmatrix} u_1 \\ u_2 \end{bmatrix} = \frac{\partial}{\partial x}\begin{bmatrix} 0.024(\partial u_1/\partial x) \\ 0.170(\partial u_2/\partial x) \end{bmatrix} + \begin{bmatrix} -F(u_1 - u_2) \\ F(u_1 - u_2) \end{bmatrix}$$

The boundary conditions on the partial derivatives of u have to be written in terms of the flux. In the form expected by pdepe, the left boundary condition is

$$\begin{bmatrix} 0 \\ u_2 \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \end{bmatrix} .* \begin{bmatrix} 0.024(\partial u_1/\partial x) \\ 0.170(\partial u_2/\partial x) \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

and the right boundary condition is

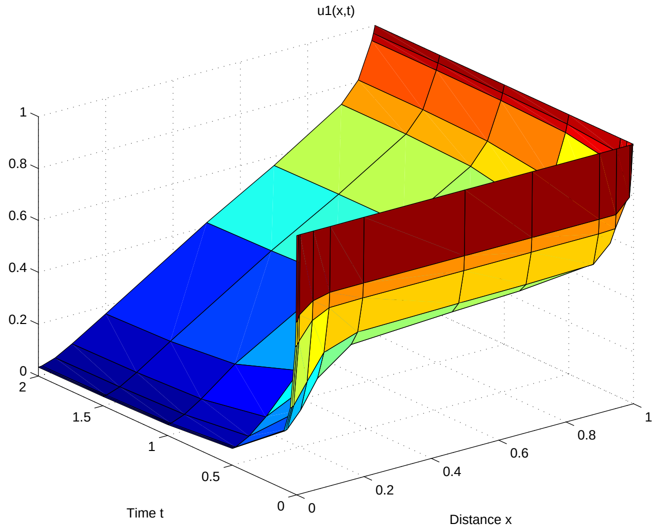
$$\begin{bmatrix} u_1 - 1 \\ 0 \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \end{bmatrix} .* \begin{bmatrix} 0.024(\partial u_1/\partial x) \\ 0.170(\partial u_2/\partial x) \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

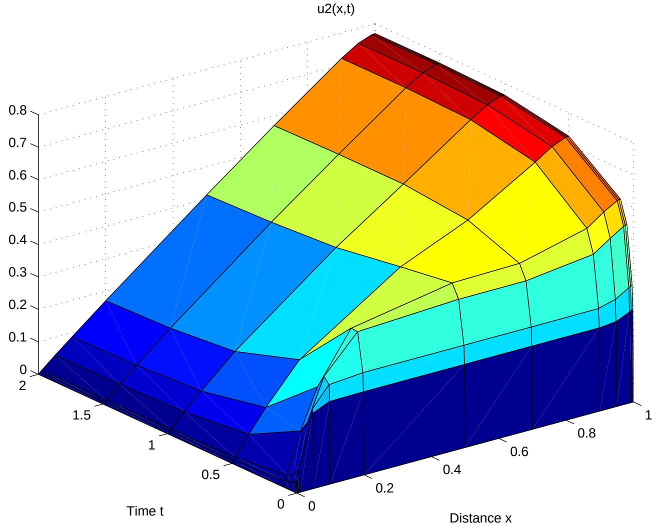
The solution changes rapidly for small t. The program selects the step size in time to resolve this sharp change, but to see this behavior in the plots, the example must select the output times accordingly. There are boundary layers in the solution at both ends of [0,1], so the example places mesh points near 0 and 1 to resolve these sharp changes. Often some experimentation is needed to select a mesh that reveals the behavior of the solution.

```
function pdex4
m = 0;
x = [0 0.005 0.01 0.05 0.1 0.2 0.5 0.7 0.9 0.95 0.99 0.995 1];
t = [0 0.005 0.01 0.05 0.1 0.5 1 1.5 2];

sol = pdepe(m,@pdex4pde,@pdex4ic,@pdex4bc,x,t);
u1 = sol(:,:,1);
u2 = sol(:,:,2);

figure
surf(x,t,u1)
title('u1(x,t)')
xlabel('Distance x')
ylabel('Time t')

figure
surf(x,t,u2)
title('u2(x,t)')
xlabel('Distance x')
ylabel('Time t')
% --------------------------------------------------------------
function [c,f,s] = pdex4pde(x,t,u,DuDx)
c = [1; 1];
f = [0.024; 0.17] .* DuDx;
y = u(1) - u(2);
F = exp(5.73*y)-exp(-11.47*y);
s = [-F; F];
% --------------------------------------------------------------
function u0 = pdex4ic(x);
u0 = [1; 0];
% --------------------------------------------------------------
function [pl,ql,pr,qr] = pdex4bc(xl,ul,xr,ur,t)
pl = [0; ul(2)];
ql = [1; 0];
pr = [ur(1)-1; 0];
qr = [0; 1];
```

In this example, the PDEs, intial conditions, and boundary conditions are coded in subfunctions pdex4pde, pdex4ic, and pdex4bc.

The surface plots show the behavior of the solution components.





See Also `function_handle`, `pdeval`, `ode15s`, `odeset`, `odeget`

References

[1] Skeel, R. D. and M. Berzins, "A Method for the Spatial Discretization of Parabolic Equations in One Space Variable," *SIAM Journal on Scientific and Statistical Computing*, Vol. 11, 1990, pp.1-32.

pdeval

Purpose

Evaluate the numerical solution of a PDE using the output of pdepe

Syntax

```
[uout,duoutdx] = pdeval(m,xmesh,ui,xout)
```

Arguments

`m` — Symmetry of the problem: slab = `0`, cylindrical = `1`, spherical = `2`. This is the first input argument used in the call to pdepe.

`xmesh` — A vector [`x0`, `x1`, ..., `xn`] specifying the points at which the elements of `ui` were computed. This is the same vector with which pdepe was called.

`ui` — A vector `sol(j,:,i)` that approximates component `i` of the solution at time t_f and mesh points `xmesh`, where `sol` is the solution returned by pdepe.

`xout` — A vector of points from the interval [`x0`,`xn`] at which the interpolated solution is requested.

Description

`[uout,duoutdx] = pdeval(m,x,ui,xout)` approximates the solution u_i and its partial derivative $\partial u_i / \partial x$ at points from the interval [`x0`,`xn`]. The `pdeval` function returns the computed values in `uout` and `duoutdx`, respectively.

Note `pdeval` evaluates the partial derivative $\partial u_i / \partial x$ rather than the flux f. Although the flux is continuous, the partial derivative may have a jump at a material interface.

See Also

`pdepe`

Purpose A sample function of two variables.

Syntax

```
Z = peaks;
Z = peaks(n);
Z = peaks(V);
Z = peaks(X,Y);

peaks;
peaks(N);
peaks(V);
peaks(X,Y);

[X,Y,Z] = peaks;
[X,Y,Z] = peaks(n);
[X,Y,Z] = peaks(V);
```

Description `peaks` is a function of two variables, obtained by translating and scaling Gaussian distributions, which is useful for demonstrating `mesh`, `surf`, `pcolor`, `contour`, and so on.

`Z = peaks;` returns a 49-by-49 matrix.

`Z = peaks(n);` returns an `n`-by-`n` matrix.

`Z = peaks(V);` returns an `n`-by-`n` matrix, where `n = length(V)`.

`Z = peaks(X,Y);` evaluates `peaks` at the given `X` and `Y` (which must be the same size) and returns a matrix the same size.

`peaks(...)` (with no output argument) plots the peaks function with `surf`.

`[X,Y,Z] = peaks(...);` returns two additional matrices, `X` and `Y`, for parametric plots, for example, `surf(X,Y,Z,del2(Z))`. If not given as input, the underlying matrices `X` and `Y` are:

```
[X,Y] = meshgrid(V,V)
```

where `V` is a given vector, or `V` is a vector of length `n` with elements equally spaced from –3 to 3. If no input argument is given, the default `n` is 49.

See Also `meshgrid`, `surf`

perms

Purpose

All possible permutations

Syntax

```
P = perms(v)
```

Description

`P = perms(v)`, where `v` is a row vector of length `n`, creates a matrix whose rows consist of all possible permutations of the `n` elements of `v`. Matrix `P` contains `n!` rows and `n` columns.

Examples

The command `perms(2:2:6)` returns *all* the permutations of the numbers 2, 4, and 6:

```
2    4    6
2    6    4
4    2    6
4    6    2
6    4    2
6    2    4
```

Limitations

This function is only practical for situations where `n` is less than about 15.

See Also

`nchoosek`, `permute`, `randperm`

Purpose Rearrange the dimensions of a multidimensional array

Syntax

```
B = permute(A,order)
```

Description `B = permute(A,order)` rearranges the dimensions of `A` so that they are in the order specified by the vector `order`. `B` has the same values of `A` but the order of the subscripts needed to access any particular element is rearranged as specified by `order`. All the elements of `order` must be unique.

Remarks `permute` and `ipermute` are a generalization of transpose (`.'`) for multidimensional arrays.

Examples Given any matrix `A`, the statement

```
permute(A,[2 1])
```

is the same as `A'`.

For example:

```
A = [1 2; 3 4]; permute(A,[2 1])
ans =
     1     3
     2     4
```

The following code permutes a three-dimensional array:

```
X = rand(12,13,14);
Y = permute(X,[2 3 1]);
size(Y)
ans =
    13    14    12
```

See Also `ipermute`

Purpose

Define persistent variable

Syntax

```
persistent X Y Z
```

Description

`persistent X Y Z` defines `X`, `Y`, and `Z` as variables that are local to the function in which they are declared yet their values are retained in memory between calls to the function. Persistent variables are similar to global variables because MATLAB creates permanent storage for both. They differ from global variables in that persistent variables are known only to the function in which they are declared. This prevents persistent variables from being changed by other functions or from the MATLAB command line.

Persistent variables are cleared when the M-file is cleared from memory or when the M-file is changed. To keep an M-file in memory until MATLAB quits, use `mlock`.

If the persistent variable does not exist the first time you issue the `persistent` statement, it is initialized to the empty matrix.

It is an error to declare a variable persistent if a variable with the same name exists in the current workspace.

Remarks

There is no function form of the persistent command (i.e., you cannot use parentheses and quote the variable names).

See Also

`clear`, `global`, `mislocked`, `mlock`, `munlock`

Purpose Ratio of a circle's circumference to its diameter, π

Syntax

```
pi
```

Description `pi` returns the floating-point number nearest the value of π. The expressions `4*atan(1)` and `imag(log(-1))` provide the same value.

Examples The expression `sin(pi)` is not exactly zero because `pi` is not exactly π.

```
sin(pi)

ans =

   1.2246e-16
```

See Also `ans`, `eps`, `i`, `Inf`, `j`, `NaN`

Purpose Pie chart

Syntax

```
pie(X)
pie(X,explode)
h = pie(...)
```

Description `pie(X)` draws a pie chart using the data in `X`. Each element in `X` is represented as a slice in the pie chart.

`pie(X,explode)` offsets a slice from the pie. `explode` is a vector or matrix of zeros and nonzeros that correspond to `X`. A non-zero value offsets the corresponding slice from the center of the pie chart, so that `X(i,j)` is offset from the center if `explode(i,j)` is nonzero. `explode` must be the same size as `X`.

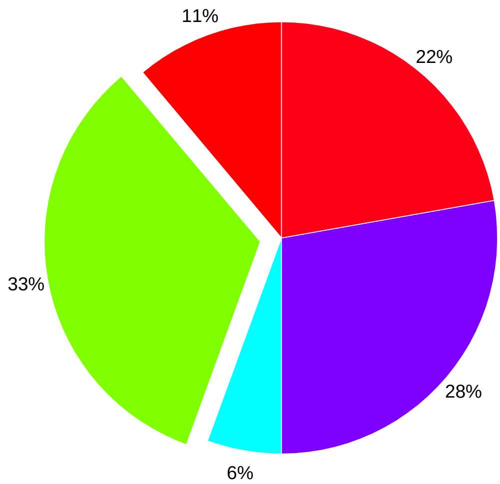
`h = pie(...)` returns a vector of handles to patch and text graphics objects.

Remarks The values in `X` are normalized via `X/sum(X)` to determine the area of each slice of the pie. If `sum(X)≤1`, the values in `X` directly specify the are of the pie slices. MATLAB draws only a partial pie if `sum(X)<1`.

Examples Emphasize the second slice in the chart by setting its corresponding `explode` element to 1.

```
x = [1 3 0.5 2.5 2];
explode = [0 1 0 0 0];
pie(x,explode)
```

```
colormap jet
```



See Also `pie3`

pie3

Purpose Three-dimensional pie chart

Syntax

```
pie3(X)
pie3(X,explode)
h = pie3(...)
```

Description `pie3(X)` draws a three-dimensional pie chart using the data in `X`. Each element in `X` is represented as a slice in the pie chart.

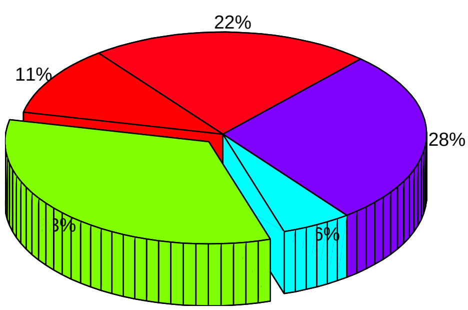
`pie3(X,explode)` specifies whether to offset a slice from the center of the pie chart. `X(i,j)` is offset from the center of the pie chart if `explode(i,j)` is nonzero. `explode` must be the same size as `X`.

`h = pie(...)` returns a vector of handles to patch, surface, and text graphics objects.

Remarks The values in `X` are normalized via `X/sum(X)` to determine the area of each slice of the pie. If `sum(X)`≤1, the values in `X` directly specify the area of the pie slices. MATLAB draws only a partial pie if `sum(X)`<1.

Examples Offset a slice in the pie chart by setting the corresponding `explode` element to 1:

```
x = [1 3 0.5 2.5 2]
explode = [0 1 0 0 0]
pie3(x,explode)
colormap hsv
```



See Also `pie`

Purpose Moore-Penrose pseudoinverse of a matrix

Syntax

```
B = pinv(A)
B = pinv(A,tol)
```

Definition The Moore-Penrose pseudoinverse is a matrix B of the same dimensions as A' satisfying four conditions:

A*B*A = A
B*A*B = B
A*B is Hermitian
B*A is Hermitian

The computation is based on svd(A) and any singular values less than tol are treated as zero.

Description B = pinv(A) returns the Moore-Penrose pseudoinverse of A.

B = pinv(A,tol) returns the Moore-Penrose pseudoinverse and overrides the default tolerance, max(size(A))*norm(A)*eps.

Examples If A is square and not singular, then pinv(A) is an expensive way to compute inv(A). If A is not square, or is square and singular, then inv(A) does not exist. In these cases, pinv(A) has some of, but not all, the properties of inv(A).

If A has more rows than columns and is not of full rank, then the overdetermined least squares problem

minimize norm(A*x-b)

does not have a unique solution. Two of the infinitely many solutions are

```
x = pinv(A)*b
```

and

```
y = A\b
```

These two are distinguished by the facts that norm(x) is smaller than the norm of any other solution and that y has the fewest possible nonzero components.

For example, the matrix generated by

```
A = magic(8); A = A(:,1:6)
```

is an 8-by-6 matrix that happens to have `rank(A) = 3`.

```
A =
    64     2     3    61    60     6
     9    55    54    12    13    51
    17    47    46    20    21    43
    40    26    27    37    36    30
    32    34    35    29    28    38
    41    23    22    44    45    19
    49    15    14    52    53    11
     8    58    59     5     4    62
```

The right-hand side is `b = 260*ones(8,1)`,

```
b =
      260
      260
      260
      260
      260
      260
      260
      260
```

The scale factor 260 is the 8-by-8 magic sum. With all eight columns, one solution to `A*x = b` would be a vector of all `1`'s. With only six columns, the equations are still consistent, so a solution exists, but it is not all `1`'s. Since the matrix is rank deficient, there are infinitely many solutions. Two of them are

```
x = pinv(A)*b
```

which is

```
x =
    1.1538
    1.4615
    1.3846
    1.3846
    1.4615
    1.1538
```

and

```
y = A\b
```

which produces this result.

```
Warning: Rank deficient, rank = 3  tol =   1.8829e-013.
y =
    4.0000
    5.0000
         0
         0
         0
   -1.0000
```

Both of these are exact solutions in the sense that `norm(A*x-b)` and `norm(A*y-b)` are on the order of roundoff error. The solution `x` is special because

```
norm(x) = 3.2817
```

is smaller than the norm of any other solution, including

```
norm(y) = 6.4807
```

On the other hand, the solution `y` is special because it has only three nonzero components.

See Also `inv`, `qr`, `rank`, `svd`

Purpose Givens plane rotation

Syntax `[G,y] = planerot(x)`

Description `[G,y] = planerot(x)` where `x` is a 2-component column vector, returns a 2-by-2 orthogonal matrix G so that `y = G*x` has `y(2) = 0`.

Examples

```
x = [3 4];
[G,y] = planerot(x')

G  =
    0.6000    0.8000
   -0.8000    0.6000

y =
     5
     0
```

See Also `qrdelete`, `qrinsert`

plot

Purpose Linear 2–D plot

Syntax

```
plot(Y)
plot(X1,Y1,...)
plot(X1,Y1,LineSpec,...)
plot(...,'PropertyName',PropertyValue,...)
h = plot(...)
```

Description `plot(Y)` plots the columns of `Y` versus their index if `Y` is a real number. If `Y` is complex, `plot(Y)` is equivalent to `plot(real(Y),imag(Y))`. In all other uses of `plot`, the imaginary component is ignored.

`plot(X1,Y1,...)` plots all lines defined by `Xn` versus `Yn` pairs. If only `Xn` or `Yn` is a matrix, the vector is plotted versus the rows or columns of the matrix, depending on whether the vector's row or column dimension matches the matrix.

`plot(X1,Y1,LineSpec,...)` plots all lines defined by the `Xn,Yn,LineSpec` triples, where `LineSpec` is a line specification that determines line type, marker symbol, and color of the plotted lines. You can mix `Xn,Yn,LineSpec` triples with `Xn,Yn` pairs: `plot(X1,Y1,X2,Y2,LineSpec,X3,Y3)`.

`plot(...,'`*`PropertyName`*`',PropertyValue,...)` sets properties to the specified property values for all line graphics objects created by `plot`. (See the "Examples" section for examples.)

`h = plot(...)` returns a column vector of handles to line graphics objects, one handle per line.

Remarks If you do not specify a color when plotting more than one line, `plot` automatically cycles through the colors in the order specified by the current axes `ColorOrder` property. After cycling through all the colors defined by `ColorOrder`, `plot` then cycles through the line styles defined in the axes `LineStyleOrder` property.

Note that, by default, MATLAB resets the `ColorOrder` and `LineStyleOrder` properties each time you call `plot`. If you want changes you make to these properties to persist, then you must define these changes as default values. For example,

```
set(0,'DefaultAxesColorOrder',[0 0 0],...
      'DefaultAxesLineStyleOrder','-|-.|--|:')
```

sets the default ColorOrder to use only the color black and sets the LineStyleOrder to use solid, dash-dot, dash-dash, and dotted line styles.

Additional Information

- See the “Creating 2-D Graphs” and “Labeling Graphs” in *Using MATLAB Graphics* for more information on plotting.
- See LineSpec for more information on specifying line styles and colors.

Examples

Specifying the Color and Size of Markers

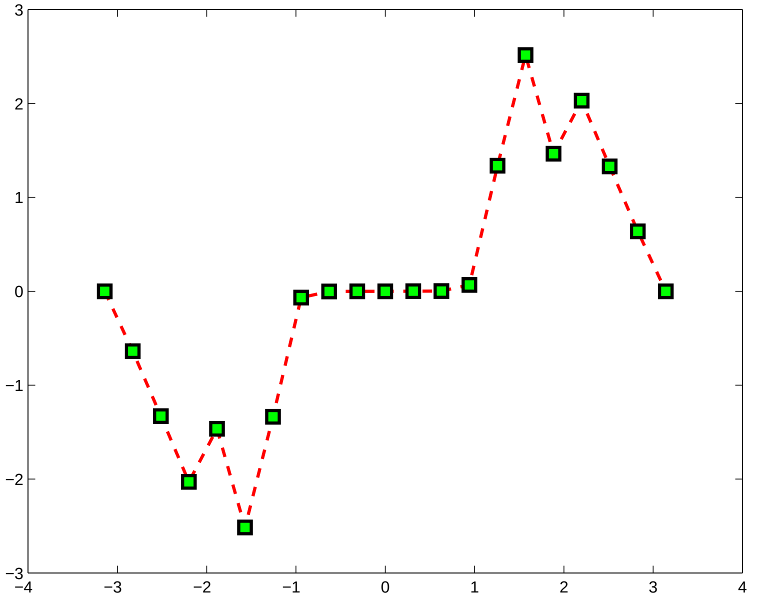
You can also specify other line characteristics using graphics properties (see line for a description of these properties):

- LineWidth – specifies the width (in points) of the line.
- MarkerEdgeColor – specifies the color of the marker or the edge color for filled markers (circle, square, diamond, pentagram, hexagram, and the four triangles).
- MarkerFaceColor – specifies the color of the face of filled markers.
- MarkerSize – specifies the size of the marker in units of points.

For example, these statements,

```
x = –pi:pi/10:pi;
y = tan(sin(x)) – sin(tan(x));
plot(x,y,'––rs','LineWidth',2,...
                'MarkerEdgeColor','k',...
                'MarkerFaceColor','g',...
                'MarkerSize',10)
```

produce this graph.

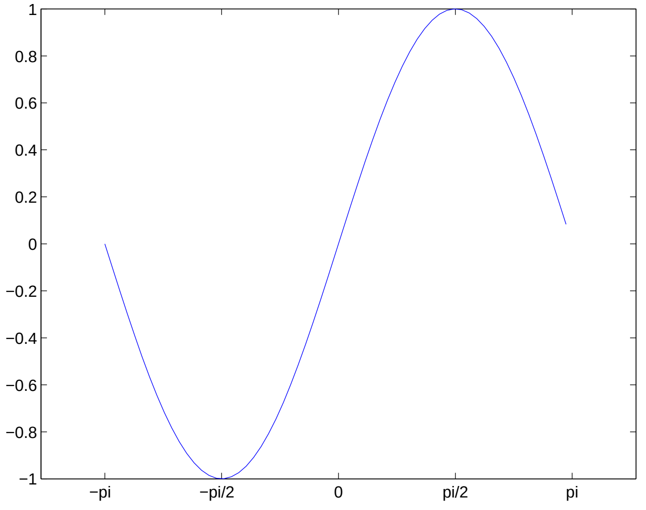


Specifying Tick Mark Location and Labeling

You can adjust the axis tick-mark locations and the labels appearing at each tick. For example, this plot of the sine function relabels the x-axis with more meaningful values,

```
x = −pi:.1:pi;
y = sin(x);
plot(x,y)
set(gca,'XTick',−pi:pi/2:pi)
set(gca,'XTickLabel',{'−pi','−pi/2','0','pi/2','pi'})
```

Now add axis labels and annotate the point –pi/4, sin(–pi/4).



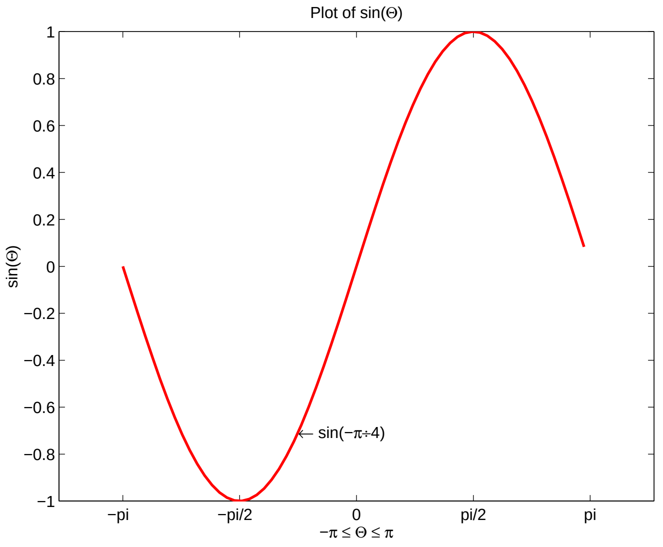
Adding Titles, Axis Labels, and Annotations

MATLAB enables you to add axis labels and titles. For example, using the graph from the previous example, add an x- and y-axis label,

```
xlabel('-\pi \leq \Theta \leq \pi')
ylabel('sin(\Theta)')
title('Plot of sin(\Theta)')
text(-pi/4,sin(-pi/4),'\leftarrow sin(-\pi\div4)',...
     'HorizontalAlignment','left')
```

Now change the line color to red by first finding the handle of the line object created by `plot` and then setting its `Color` property. In the same statement, set the `LineWidth` property to 2 points.

```
set(findobj(gca,'Type','line','Color',[0 0 1]),...
    'Color','red',...
    'LineWidth',2)
```



See Also

`axis`, `bar`, `grid`, `legend`, `line`, `LineSpec`, `loglog`, `plotyy`, `semilogx`, `semilogy`, `subplot`, `xlabel`, `xlim`, `ylabel`, `ylim`, `zlabel`, `zlim`, `stem`

See the `text` `String` property for a list of symbols and how to display them.

See `plotedit` for information on using the plot annotation tools in the figure window toolbar.

Purpose

Linear 3-D plot

Syntax

```
plot3(X1,Y1,Z1,...)
plot3(X1,Y1,Z1,LineSpec,...)
plot3(...,'PropertyName',PropertyValue,...)
h = plot3(...)
```

Description

The `plot3` function displays a three-dimensional plot of a set of data points.

`plot3(X1,Y1,Z1,...)`, where `X1`, `Y1`, `Z1` are vectors or matrices, plots one or more lines in three-dimensional space through the points whose coordinates are the elements of `X1`, `Y1`, and `Z1`.

`plot3(X1,Y1,Z1,LineSpec,...)` creates and displays all lines defined by the `Xn,Yn,Zn,LineSpec` quads, where `LineSpec` is a line specification that determines line style, marker symbol, and color of the plotted lines.

`plot3(...,'`*`PropertyName`*`',PropertyValue,...)` sets properties to the specified property values for all Line graphics objects created by `plot3`.

`h = plot3(...)` returns a column vector of handles to line graphics objects, with one handle per line.

Remarks

If one or more of `X1`, `Y1`, `Z1` is a vector, the vectors are plotted versus the rows or columns of the matrix, depending whether the vectors' lengths equal the number of rows or the number of columns.

You can mix `Xn,Yn,Zn` triples with `Xn,Yn,Zn,`*`LineSpec`* quads, for example,

```
plot3(X1,Y1,Z1,X2,Y2,Z2,LineSpec,X3,Y3,Z3)
```

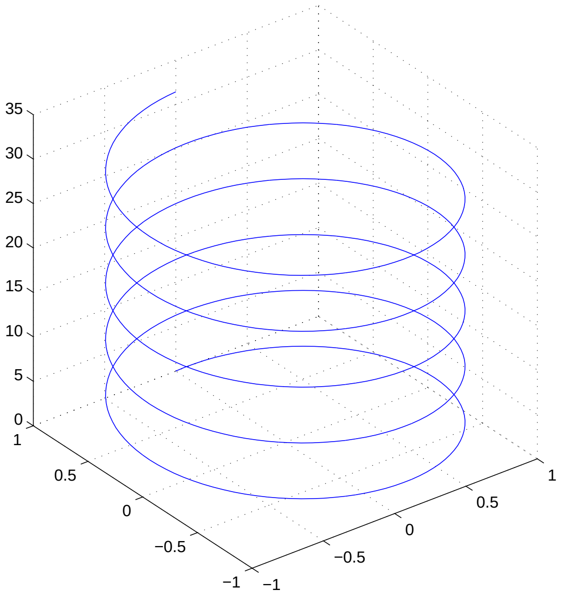
See `LineSpec` and `plot` for information on line types and markers.

Examples

Plot a three-dimensional helix.

```
t = 0:pi/50:10*pi;
plot3(sin(t),cos(t),t)
grid on
```

```
axis square
```



See Also `axis`, `bar3`, `grid`, `line`, `LineSpec`, `loglog`, `plot`, `semilogx`, `semilogy`, `subplot`

Purpose

Start plot edit mode to allow editing and annotation of plots

Syntax

```
plotedit on
plotedit off
plotedit
plotedit('state')
plotedit(h)
plotedit(h,'state')
```

Description

`plotedit on` starts plot edit mode for the current figure, allowing you to use a graphical interface to annotate and edit plots easily. In plot edit mode, you can label axes, chang line styles, and adding text, line, and arrow annotations.

`plotedit off` ends plot mode for the current figure.

`plotedit` toggles the plot edit mode for the current figure.

`plotedit(h)` toggles the plot edit mode for the figure specified by figure handle `h`.

`plotedit('state')` specifies the `plotedit` state for the current figure. Values for `state` can be as shown.

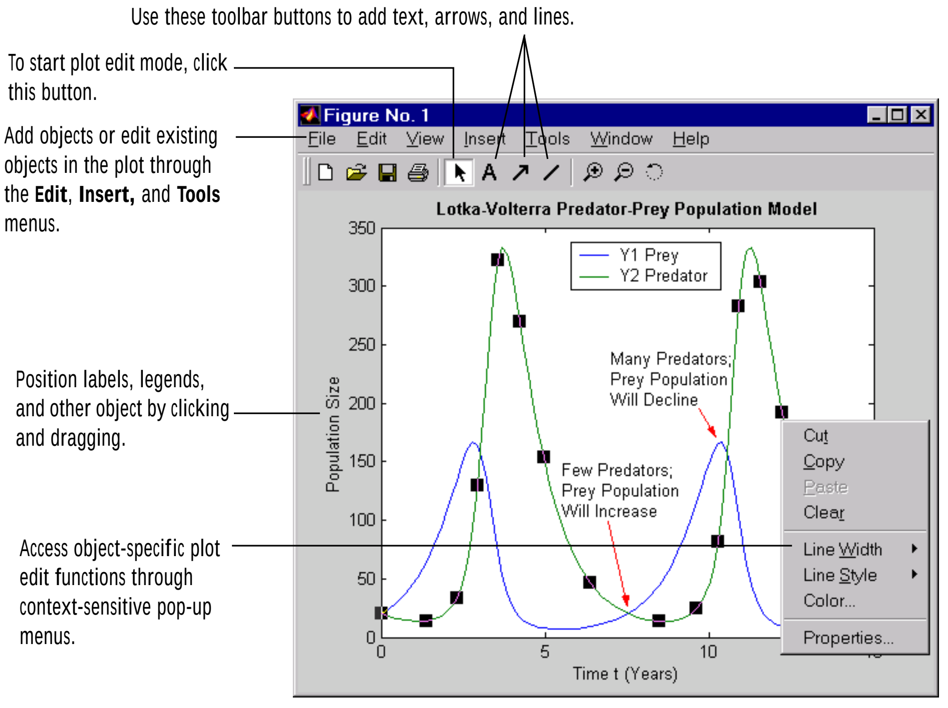
Value for state	Description
`on`	Starts plot edit mode
`off`	Ends plot edit mode
`showtoolsmenu`	Displays the **Tools** menu in the menu bar
`hidetoolsmenu`	Removes the **Tools** menu from the menu bar

Note `hidetoolsmenu` is intended for GUI developers who do not want the **Tools** menu to appear in applications that use the figure window.

`plotedit(h,'state')` specifies the `plotedit` `state` for figure handle `h`.

Remarks

Plot Editing Mode Graphical Interface Components



Help

For more information about editing plots, select **Plot Editing** from the Figure window **Help** menu. For help with other MATLAB graphics features, select **Creating Plots**.

Examples

Start plot edit mode for figure 2:

```
plotedit(2)
```

End plot edit mode for figure 2:

```
plotedit(2, 'off')
```

Hide the **Tools** menu for the current figure:

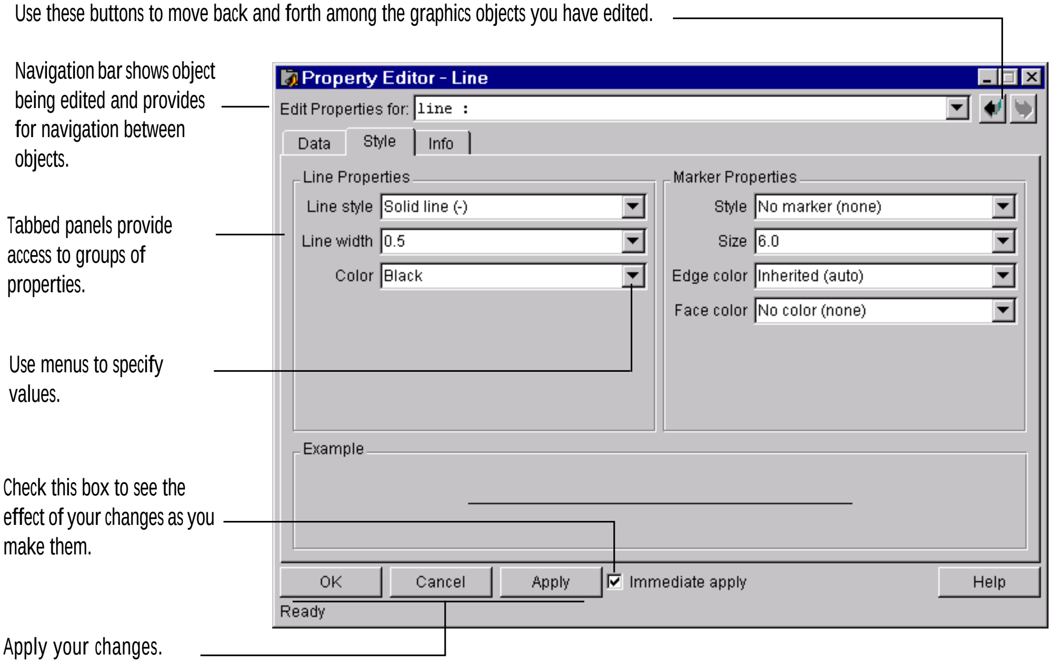
```
plotedit('hidetoolsmenu')
```

See Also

`axes`, `line`, `open`, `plot`, `print`, `saveas`, `text`, `propedit`

Remarks

Property Editor Graphical User Interface Components



See Also

`plotedit`

plotmatrix

Purpose Draw scatter plots

Syntax

```
plotmatrix(X,Y)
plotmatrix(...,'LineSpec')
[H,AX,BigAx,P] = plotmatrix(...)
```

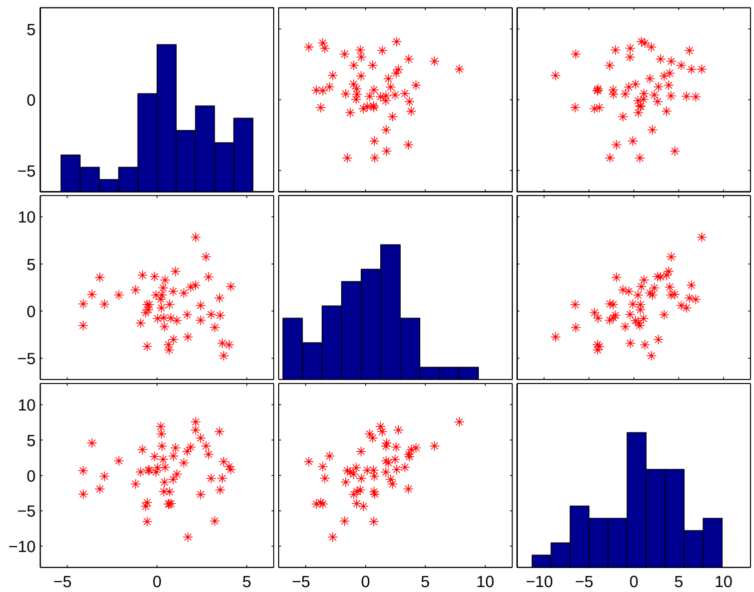
Description `plotmatrix(X,Y)` scatter plots the columns of `X` against the columns of `Y`. If `X` is *p*-by-*m* and `Y` is *p*-by-*n*, `plotmatrix` produces an *n*-by-*m* matrix of axes. `plotmatrix(Y)` is the same as `plotmatrix(Y,Y)` except that the diagonal is replaced by `hist(Y(:,i))`.

`plotmatrix(...,'LineSpec')` uses a `LineSpec` to create the scatter plot.The default is `'.'` .

`[H,AX,BigAx,P] = plotmatrix(...)` returns a matrix of handles to the objects created in `H`, a matrix of handles to the individual subaxes in `AX`, a handle to a big (invisible) axes that frames the subaxes in `BigAx`, and a matrix of handles for the histogram plots in `P`. `BigAx` is left as the current axes so that a subsequent `title`, `xlabel`, or `ylabel` commands are centered with respect to the matrix of axes.

Examples Generate plots of random data.

```
x = randn(50,3); y = x*[-1 2 1;2 0 1;1 -2 3;]';
plotmatrix(y,'*r')
```



See Also `scatter`, `scatter3`

plotyy

Purpose Create graphs with y axes on both left and right side

Syntax

```
plotyy(X1,Y1,X2,Y2)
plotyy(X1,Y1,X2,Y2,'function')
plotyy(X1,Y1,X2,Y2,'function1','function2')
[AX,H1,H2] = plotyy(...)
```

Description

`plotyy(X1,Y1,X2,Y2)` plots `X1` versus `Y1` with y-axis labeling on the left and plots `X2` versus `Y2` with y-axis labeling on the right.

`plotyy(X1,Y1,X2,Y2,'function')` uses the plotting function specified by the string '`function`' instead of `plot` to produce each graph. '`function`' can be `plot`, `semilogx`, `semilogy`, `loglog`, `stem` or any MATLAB function that accepts the syntax:

```
h = function(x,y)
```

`plotyy(X1,Y1,X2,Y2,'function1','function2')` uses `function1(X1,Y1)` to plot the data for the left axis and `function2(X2,Y2)` to plot the data for the right axis.

`[AX,H1,H2] = plotyy(...)` returns the handles of the two axes created in `AX` and the handles of the graphics objects from each plot in `H1` and `H2`. `AX(1)` is the left axes and `AX(2)` is the right axes.

Examples

This example graphs two mathematical functions using `plot` as the plotting function. The two y-axes enable you to display both sets of data on one graph even though relative values of the data are quite different.

```
x = 0:0.01:20;
y1 = 200*exp(-0.05*x).*sin(x);
y2 = 0.8*exp(-0.5*x).*sin(10*x);
[AX,H1,H2] = plotyy(x,y1,x,y2,'plot');
```

You can use the handles returned by `plotyy` to label the axes and set the line styles used for plotting. With the axes handles you can specify the `YLabel` properties of the left- and right-side y-axis:

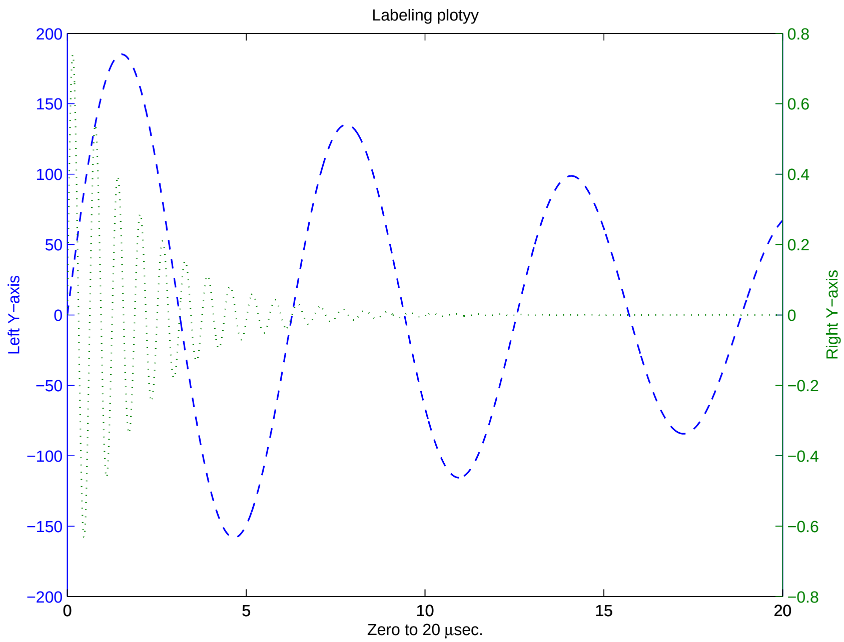
```
set(get(AX(1),'Ylabel'),'String','Left Y-axis')
set(get(AX(2),'Ylabel'),'String','Right Y-axis')
```

Use the `xlabel` and `title` commands to label the x-axis and add a title:

```
xlabel('Zero to 20 \musec.')
title('Labeling plotyy')
```

Use the line handles to set the `LineStyle` properties of the left- and right-side plots:

```
set(H1,'LineStyle','--')
set(H2,'LineStyle',':')
```



See Also

`plot`, `loglog`, `semilogx`, `semilogy`, axes properties: `XAxisLocation`, `YAxisLocation`

The axes chapter in the *Using MATLAB Graphics* manual for information on multi-axis axes.

pol2cart

Purpose Transform polar or cylindrical coordinates to Cartesian

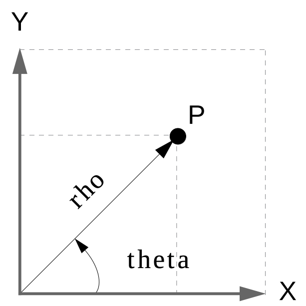
Syntax

```
[X,Y] = pol2cart(THETA,RHO)
[X,Y,Z] = pol2cart(THETA,RHO,Z)
```

Description `[X,Y] = pol2cart(THETA,RHO)` transforms the polar coordinate data stored in corresponding elements of `THETA` and `RHO` to two-dimensional Cartesian, or *xy*, coordinates. The arrays `THETA` and `RHO` must be the same size (or either can be scalar). The values in `THETA` must be in radians.

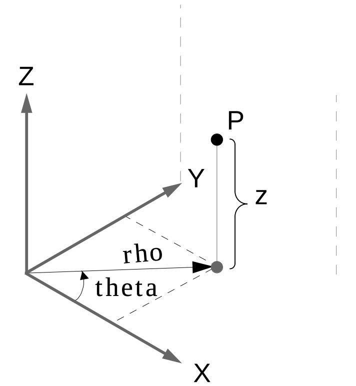
`[X,Y,Z] = pol2cart(THETA,RHO,Z)` transforms the cylindrical coordinate data stored in corresponding elements of `THETA`, `RHO`, and `Z` to three-dimensional Cartesian, or *xyz*, coordinates. The arrays `THETA` , `RHO`, and `Z` must be the same size (or any can be scalar). The values in `THETA` must be in radians.

Algorithm The mapping from polar and cylindrical coordinates to Cartesian coordinates is:



Polar to Cartesian Mapping

```
theta = atan2(y,x)
rho = sqrt(x.^2 + y.^2)
```



Cylindrical to Cartesian Mapping

```
theta = atan2(y,x)
rho = sqrt(x.^2 + y.^2)
z = z
```

See Also `cart2pol`, `cart2sph`, `sph2cart`

Purpose Plot polar coordinates

Syntax

```
polar(theta,rho)
polar(theta,rho,LineSpec)
```

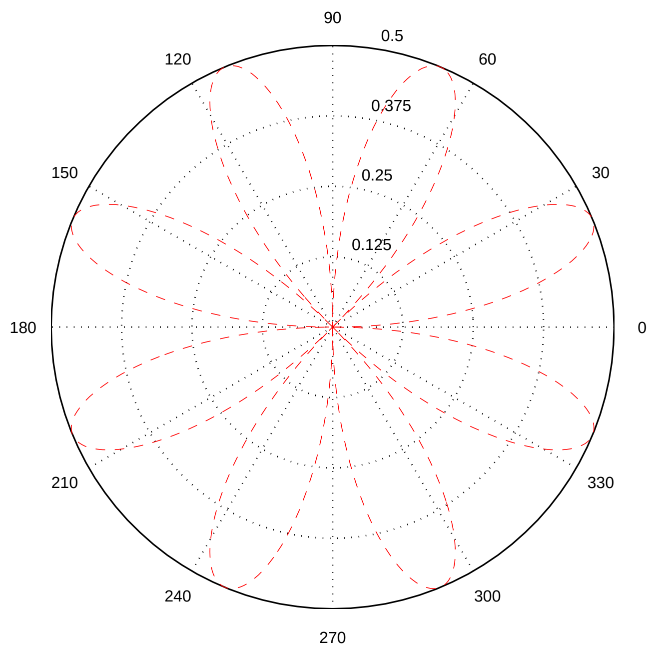
Description The `polar` function accepts polar coordinates, plots them in a Cartesian plane, and draws the polar grid on the plane.

`polar(theta,rho)` creates a polar coordinate plot of the angle `theta` versus the radius `rho`. `theta` is the angle from the *x*-axis to the radius vector specified in radians; `rho` is the length of the radius vector specified in dataspace units.

`polar(theta,rho,LineSpec)` `LineSpec` specifies the line type, plot symbol, and color for the lines drawn in the polar plot.

Examples

```
Create a simple polar plot using a dashed, red line:
t = 0:.01:2*pi;
polar(t,sin(2*t).*cos(2*t),'--r')
```



See Also `cart2pol`, `compass`, `LineSpec`, `plot`, `pol2cart`, `rose`

Purpose

Polynomial with specified roots

Syntax

```
p = poly(A)
p = poly(r)
```

Description

`p = poly(A)` where `A` is an `n`-by-`n` matrix returns an `n+1` element row vector whose elements are the coefficients of the characteristic polynomial, $\det(sI - A)$. The coefficients are ordered in descending powers: if a vector `c` has `n+1` components, the polynomial it represents is $c_1 s^n + \ldots + c_n s + c_{n+1}$

`p = poly(r)` where `r` is a vector returns a row vector whose elements are the coefficients of the polynomial whose roots are the elements of `r`.

Remarks

Note the relationship of this command to

```
r = roots(p)
```

which returns a column vector whose elements are the roots of the polynomial specified by the coefficients row vector `p`. For vectors, `roots` and `poly` are inverse functions of each other, up to ordering, scaling, and roundoff error.

Examples

MATLAB displays polynomials as row vectors containing the coefficients ordered by descending powers. The characteristic equation of the matrix

```
A =

    1    2    3
    4    5    6
    7    8    0
```

is returned in a row vector by `poly`:

```
p = poly(A)

p =
    1    -6    -72    -27
```

The roots of this polynomial (eigenvalues of matrix `A`) are returned in a column vector by `roots`:

```
r = roots(p)
```

```
r =

    12.1229
    -5.7345
    -0.3884
```

Algorithm

The algorithms employed for `poly` and `roots` illustrate an interesting aspect of the modern approach to eigenvalue computation. `poly(A)` generates the characteristic polynomial of `A`, and `roots(poly(A))` finds the roots of that polynomial, which are the eigenvalues of `A`. But both `poly` and `roots` use `eig`, which is based on similarity transformations. The classical approach, which characterizes eigenvalues as roots of the characteristic polynomial, is actually reversed.

If `A` is an `n`-by-`n` matrix, `poly(A)` produces the coefficients `c(1)` through `c(n+1)`, with `c(1) = 1`, in

$$det(\lambda I - A) = c_1\lambda^n + \ldots + c_n\lambda + c_{n+1}$$

The algorithm is

```
z = eig(A);
c = zeros(n+1,1); c(1) = 1;
for j = 1:n
    c(2:j+1) = c(2:j+1)-z(j)*c(1:j);
end
```

This recursion is easily derived by expanding the product.

$$(\lambda - \lambda_1)(\lambda - \lambda_2)\ldots(\lambda - \lambda_n)$$

It is possible to prove that `poly(A)` produces the coefficients in the characteristic polynomial of a matrix within roundoff error of `A`. This is true even if the eigenvalues of `A` are badly conditioned. The traditional algorithms for obtaining the characteristic polynomial, which do not use the eigenvalues, do not have such satisfactory numerical properties.

See Also

`conv`, `polyval`, `residue`, `roots`

Purpose Area of polygon

Syntax

```
A = polyarea(X,Y)
A = polyarea(X,Y,dim)
```

Description

`A = polyarea(X,Y)` returns the area of the polygon specified by the vertices in the vectors `X` and `Y`.

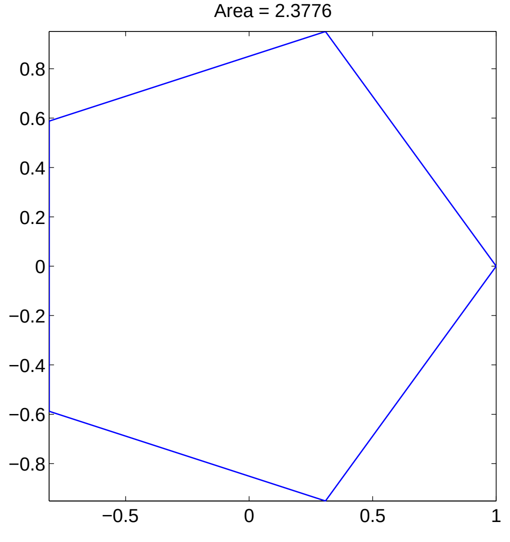
If `X` and `Y` are matrices of the same size, then `polyarea` returns the area of polygons defined by the columns `X` and `Y`.

If `X` and `Y` are multidimensional arrays, `polyarea` returns the area of the polygons in the first nonsingleton dimension of `X` and `Y`.

`A = polyarea(X,Y,dim)` operates along the dimension specified by scalar `dim`.

Examples

```
L = linspace(0,2.*pi,6); xv = cos(L)';yv = sin(L)';
xv = [xv ; xv(1)]; yv = [yv ; yv(1)];
A = polyarea(xv,yv);
plot(xv,yv); title(['Area = ' num2str(A)]); axis image
```



See Also `convhull`, `inpolygon`, `rectint`

polyder

Purpose Polynomial derivative

Syntax

```
k = polyder(p)
k = polyder(a,b)
[q,d] = polyder(b,a)
```

Description The `polyder` function calculates the derivative of polynomials, polynomial products, and polynomial quotients. The operands `a`, `b`, and `p` are vectors whose elements are the coefficients of a polynomial in descending powers.

`k = polyder(p)` returns the derivative of the polynomial `p`.

`k = polyder(a,b)` returns the derivative of the product of the polynomials `a` and `b`.

`[q,d] = polyder(b,a)` returns the numerator `q` and denominator `d` of the derivative of the polynomial quotient `b/a`.

Examples The derivative of the product

$$(3x^2 + 6x + 9)(x^2 + 2x)$$

is obtained with

```
a = [3 6 9];
b = [1 2 0];
k = polyder(a,b)
k =
    12    36    42    18
```

This result represents the polynomial

$$12x^3 + 36x^2 + 42x + 18$$

See Also `conv`, `deconv`

Purpose Polynomial eigenvalue problem

Syntax

```
[X,e] = polyeig(A0,A1,...Ap)
e = polyeig(A0,A1,..,Ap)
```

Description `[X,e] = polyeig(A0,A1,...Ap)` solves the polynomial eigenvalue problem of degree `p`

$$(A_0 + \lambda A_1 + \ldots + \lambda^P A_p)x = 0$$

where polynomial degree `p` is a non-negative integer, and `A0,A1,...Ap` are input matrices of order `n`. Output matrix `X`, of size `n`-by-`n*p`, contains eigenvectors in its columns. Output vector `e`, of length `n*p`, contains eigenvalues.

If `lambda` is the `j`th eigenvalue in `e`, and `x` is the `j`th column of eigenvectors in `X`, then `(A0 + lambda*A1 + ... + lambda^p*Ap)*x` is approximately `0`.

`e = polyeig(A0,A1,..,Ap)` is a vector of length `n*p` whose elements are the eigenvalues of the polynomial eigenvalue problem.

Remarks Based on the values of `p` and `n`, `polyeig` handles several special cases:

- `p = 0`, or `polyeig(A)` is the standard eigenvalue problem: `eig(A)`.
- `p = 1`, or `polyeig(A,B)` is the generalized eigenvalue problem: `eig(A,-B)`.
- `n = 1`, or `polyeig(a0,a1,...ap)` for scalars `a0`, `a1` ..., `ap` is the standard polynomial problem: `roots([ap ... a1 a0])`.

Algorithm If both `A0` and `Ap` are singular, the problem is potentially ill posed; solutions might not exist or they might not be unique. In this case, the computed solutions may be inaccurate. `polyeig` attempts to detect this situation and display an appropriate warning message. If either one, but not both, of `A0` and `Ap` is singular, the problem is well posed but some of the eigenvalues may be zero or infinite (`Inf`).

The `polyeig` function uses the QZ factorization to find intermediate results in the computation of generalized eigenvalues. It uses these intermediate results to determine if the eigenvalues are well-determined. See the descriptions of `eig` and `qz` for more on this.

See Also

eig, qz

Purpose Polynomial curve fitting

Syntax

```
p = polyfit(x,y,n)
[p,S] = polyfit(x,y,n)
[p,S,mu] = polyfit(x,y,n)
```

Description `p = polyfit(x,y,n)` finds the coefficients of a polynomial `p(x)` of degree `n` that fits the data, `p(x(i))` to `y(i)`, in a least squares sense. The result `p` is a row vector of length `n+1` containing the polynomial coefficients in descending powers

$$p(x) = p_1 x^n + p_2 x^{n-1} + \ldots + p_n x + p_{n+1}$$

`[p,S] = polyfit(x,y,n)` returns the polynomial coefficients `p` and a structure `S` for use with `polyval` to obtain error estimates or predictions. If the errors in the data `y` are independent normal with constant variance, `polyval` produces error bounds that contain at least 50% of the predictions.

`[p,S,mu] = polyfit(x,y,n)` finds the coefficients of a polynomial in

$$\hat{x} = \frac{x - \mu_1}{\mu_2}$$

where $\mu_1 = \mathrm{mean}(x)$ and $\mu_2 = \mathrm{std}(x)$. `mu` is the two-element vector $[\mu_1, \mu_2]$. This centering and scaling transformation improves the numerical properties of both the polynomial and the fitting algorithm.

Examples This example involves fitting the error function, `erf(x)`, by a polynomial in `x`. This is a risky project because `erf(x)` is a bounded function, while polynomials are unbounded, so the fit might not be very good.

First generate a vector of `x` points, equally spaced in the interval $[0, 2.5]$; then evaluate `erf(x)` at those points.

```
x = (0: 0.1: 2.5)';
y = erf(x);
```

The coefficients in the approximating polynomial of degree 6 are

```
p = polyfit(x,y,6)
```

```
p =

 0.0084  -0.0983   0.4217   -0.7435  0.1471   1.1064  0.0004
```

There are seven coefficients and the polynomial is

$$0.0084x^6 - 0.0983x^5 + 0.4217x^4 - 0.7435x^3 + 0.1471x^2 + 1.1064x + 0.0004$$

To see how good the fit is, evaluate the polynomial at the data points with

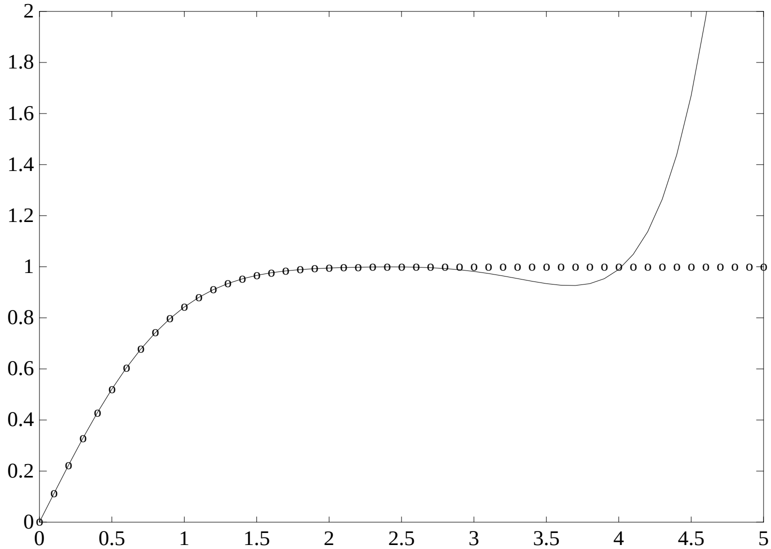
```
f = polyval(p,x);
```

A table showing the data, fit, and error is

```
 table = [x y f y-f]

 table =

    0          0          0.0004    -0.0004
    0.1000     0.1125     0.1119     0.0006
    0.2000     0.2227     0.2223     0.0004
    0.3000     0.3286     0.3287    -0.0001
    0.4000     0.4284     0.4288    -0.0004
    ...
    2.1000     0.9970     0.9969     0.0001
    2.2000     0.9981     0.9982    -0.0001
    2.3000     0.9989     0.9991    -0.0003
    2.4000     0.9993     0.9995    -0.0002
    2.5000     0.9996     0.9994     0.0002
```

So, on this interval, the fit is good to between three and four digits. Beyond this interval the graph shows that the polynomial behavior takes over and the approximation quickly deteriorates.

```
x = (0: 0.1: 5)';
y = erf(x);
f = polyval(p,x);
plot(x,y,'o',x,f,'-')
axis([0  5  0  2])
```



Algorithm

The `polyfit` M-file forms the Vandermonde matrix, V, whose elements are powers of x.

$$v_{i,\,j} = x_i^{n-j}$$

It then uses the backslash operator, \, to solve the least squares problem

$$V_p \cong y$$

You can modify the M-file to use other functions of x as the basis functions.

See Also

`poly`, `polyval`, `roots`

polyint

Purpose Integrate polynomial analytically

Syntax

```
polyint(p,k)
polyint(p)
```

Description `polyint(p,k)` returns a polynomial representing the integral of polynomial `p`, using a scalar constant of integration `k`.

`polyint(p)` assumes a constant of integration `k=0`.

See Also `polyder`, `polyval`, `polyvalm`, `polyfit`

Purpose Polynomial evaluation

Syntax

```
y = polyval(p,x)
y = polyval(p,x,[],mu)
[y,delta] = polyval(p,x,S)
[y,delta] = polyval(p,x,S,mu)
```

Description `y = polyval(p,x)` returns the value of a polynomial of degree `n` evaluated at `x`. The input argument `p` is a vector of length `n+1` whose elements are the coefficients in descending powers of the polynomial to be evaluated.

$$y = p_1x^n + p_2x^{n-1} + \ldots + p_nx + p_{n+1}$$

`x` can be a matrix or a vector. In either case, `polyval` evaluates `p` at each element of `x`.

`y = polyval(p,x,[],mu)` uses $\hat{x} = (x - \mu_1)/\mu_2$ in place of x. In this equation, $\mu_1 = \mathrm{mean}(x)$ and $\mu_2 = \mathrm{std}(x)$. The centering and scaling parameters `mu` = $[\mu_1, \mu_2]$ are optional output computed by `polyfit`.

`[y,delta] = polyval(p,x,S)` and `[y,delta] = polyval(p,x,S,mu)` use the optional output structure `S` generated by `polyfit` to generate error estimates, `y±delta`. If the errors in the data input to `polyfit` are independent normal with constant variance, `y±delta` contains at least 50% of the predictions.

Remarks The `polyvalm(p,x)` function, with `x` a matrix, evaluates the polynomial in a matrix sense. See `polyvalm` for more information.

Examples The polynomial $p(x) = 3x^2 + 2x + 1$ is evaluated at $x = 5, 7,$ and 9 with

```
p = [3 2 1];
polyval(p,[5 7 9])
```

which results in

```
ans =

    86   162   262
```

For another example, see `polyfit`.

See Also

polyfit, polyvalm

Purpose Matrix polynomial evaluation

Syntax

```
Y = polyvalm(p,X)
```

Description `Y = polyvalm(p,X)` evaluates a polynomial in a matrix sense. This is the same as substituting matrix `X` in the polynomial `p`.

Polynomial `p` is a vector whose elements are the coefficients of a polynomial in descending powers, and `X` must be a square matrix.

Examples The Pascal matrices are formed from Pascal's triangle of binomial coefficients. Here is the Pascal matrix of order 4.

```
X = pascal(4)
X =
    1    1    1    1
    1    2    3    4
    1    3    6   10
    1    4   10   20
```

Its characteristic polynomial can be generated with the `poly` function.

```
p = poly(X)
p =
   1    -29    72    -29    1
```

This represents the polynomial $x^4 - 29x^3 + 72x^2 - 29x + 1$.

Pascal matrices have the curious property that the vector of coefficients of the characteristic polynomial is palindromic; it is the same forward and backward.

Evaluating this polynomial at each element is not very interesting.

```
polyval(p,X)
ans =
   16      16      16      16
   16      15    -140    -563
   16    -140   -2549  -12089
   16    -563  -12089  -43779
```

But evaluating it in a matrix sense is interesting.

```
polyvalm(p,X)
```

```
ans =
     0     0     0     0
     0     0     0     0
     0     0     0     0
     0     0     0     0
```

The result is the zero matrix. This is an instance of the Cayley-Hamilton theorem: a matrix satisfies its own characteristic equation.

See Also polyfit, polyval

Purpose Base 2 power and scale floating-point numbers

Syntax

```
X = pow2(Y)
X = pow2(F,E)
```

Description `X = pow2(Y)` returns an array `X` whose elements are 2 raised to the power `Y`.

`X = pow2(F,E)` computes $x = f * 2^e$ for corresponding elements of `F` and `E`. The result is computed quickly by simply adding `E` to the floating-point exponent of `F`. Arguments `F` and `E` are real and integer arrays, respectively.

Remarks This function corresponds to the ANSI C function `ldexp()` and the IEEE floating-point standard function `scalbn()`.

Examples For IEEE arithmetic, the statement `X = pow2(F,E)` yields the values:

F	E	X
1/2	1	1
pi/4	2	pi
-3/4	2	-3
1/2	-51	eps
1-eps/2	1024	realmax
1/2	-1021	realmin

See Also `log2`, `exp`, `hex2num`, `realmax`, `realmin`

The arithmetic operators `^` and `.^`

ppval

Purpose Evaluate piecewise polynomial.

Syntax

```
v = ppval(pp,xx)
v = ppval(xx,pp)
```

Description `v = ppval(pp,xx)` returns the value at the points `xx` of the piecewise polynomial contained in `pp`, as constructed by `spline` or the spline utility `mkpp`.

`v = ppval(xx,pp)` returns the same result but can be used with functions like `fminbnd`, `fzero` and `quad` that take a function as an argument.

Examples Compare the results of integrating the function `cos`

```
a = 0; b = 10;
int1 = quad(@cos,a,b,[],[])

int1 =
   -0.5440
```

with the results of integrating the piecewise polynomial `pp` that approximates the cosine function by interpolating the computed values `x` and `y`.

```
x = a:b;
y = cos(x);
pp = spline(x,y);
int2 = quad(@ppval,a,b,[],[],pp)

int2 =
   -0.5485
```

`int1` provides the integral of the cosine function over the interval `[a,b]`, while `int2` provides the integral over the same interval of the piecewise polynomial `pp`.

See Also `mkpp`, `spline`, `unmkpp`

Purpose Generate list of prime numbers

Syntax

```
p = primes(n)
```

Description

`p = primes(n)` returns a row vector of the prime numbers less than or equal to `n`. A prime number is one that has no factors other than `1` and itself.

Examples

```
p = primes(37)

p =

     2     3     5     7    11    13    17    19    23    29    31    37
```

See Also `factor`

print, printopt

Purpose Create hardcopy output

Syntax

```
print
print -device -options filename
[pcmd,dev] = printopt
```

Description

`print` and `printopt` produce hardcopy output. All arguments to the `print` command are optional. You can use them in any combination or order.

`print` sends the contents of the current figure, including bitmap representations of any user interface controls, to the printer using the device and system printing command defined by `printopt`.

`print -`*`device`* specifies a print driver (such as color PostScript) or a graphics-file format (such as TIFF). If the `-device` is set to `-dmeta` or `-dbitmap` (Windows only), the figure is saved to the clipboard. If you omit `-device`, print uses the default value stored by `printopt`. The Devices section lists all supported device types.

`print -`*`options`* specifies print options that modify the action of the `print` command. (For example, the `–noui` option suppresses printing of user interface controls.) The Options section lists available options.

`print filename` directs the output to the file designated by `filename`. If `filename` does not include an extension, `print` appends an appropriate extension, depending on the driver or format specified (e.g., `.ps` or `.tif`).

`print(...)` is the function form of print. It enables you to pass variables for any input arguments. This form is useful passing filenames and handles. See Batch Processing for an example.

`[pcmd,dev] = printopt` returns strings containing the current system-dependent printing command and output device. `printopt` is an M-file used by `print` to produce the hardcopy output. You can edit the M-file `printopt.m` to set your default printer type and destination.

`pcmd` and `dev` are platform-dependent strings. `pcmd` contains the command that `print` uses to send a file to the printer. `dev` contains the printer driver or graphics format option for the `print` command. Their defaults are platform dependent.

Platform	System Printing Command	Driver or Format
UNIX	`lpr –r –s`	`–dps2`
Windows	`COPY /B %s LPT1:`	`–dwin`

Drivers

The table below shows the complete list of printer drivers supported by MATLAB. If you do not specify a driver, MATLAB uses the default setting shown in the previous table.

Some of the drivers are available from a product called Ghostscript, which is shipped with MATLAB. The last column indicates when Ghostscript is used.

Some drivers are not available on all platforms. This is noted in the first column of the table.

Printer Driver	MATLAB call	Ghost-Script
Canon BubbleJet BJ10e	`print -dbj10e`	Yes
Canon BubbleJet BJ200 color	`print -dbj200`	Yes
Canon Color BubbleJet BJC-70/BJC-600/BJC-4000	`print -dbjc600`	Yes
Canon Color BubbleJet BJC-800	`print -dbjc800`	Yes
DEC LN03	`print -dln03`	Yes
Epson and compatible 9- or 24-pin dot matrix print drivers	`print -depson`	Yes
Epson and compatible 9-pin with interleaved lines (triple resolution)	`print -deps9high`	Yes
Epson LQ-2550 and compatible; color (not supported on HP-700)	`print -depsonc`	Yes
Fujitsu 3400/2400/1200	`print -depsonc`	Yes

Printer Driver	MATLAB call	Ghost-Script
HP DesignJet 650C color (not supported on Windows or DEC Alpha)	`print -ddnj650c`	Yes
HP DeskJet 500	`print -ddjet500`	Yes
HP DeskJet 500C (creates black-and-white output)	`print -dcdjmono`	Yes
HP DeskJet 500C (with 24 bit/pixel color and high-quality Floyd-Steinberg color dithering) (not supported on Windows or DEC Alpha)	`print -dcdjcolor`	Yes
HP DeskJet 500C/540C color (not supported on Windows or DEC Alpha)	`print -dcdj500`	Yes
HP Deskjet 550C color (not supported on Windows or DEC Alpha)	`print -dcdj550`	Yes
HP DeskJet and **DeskJet Plus**	`print -ddeskjet`	Yes
HP LaserJet	`print -dlaserjet`	Yes
HP LaserJet+	`print -dljetplus`	Yes
HP LaserJet IIP	`print -dljet2p`	Yes
HP LaserJet III	`print -dljet3`	Yes
HP LaserJet 4.5L and 5P	`print -dljet4`	Yes
HP LaserJet 5 and 6	`print -dpxlmono`	Yes
HP PaintJet color	`print -dpaintjet`	Yes
HP PaintJet XL color	`print -dpjxl`	Yes
HP PaintJet XL color	`print -dpjetxl`	Yes
HP PaintJet XL300 color (not supported on Windows or DEC Alpha)	`print -dpjxl300`	Yes
HPGL for HP 7475A and other compatible plotters. (Renderer cannot be set to Z-buffer.)	`print -dhpgl`	Yes

Printer Driver	MATLAB call	Ghost-Script
IBM 9-pin Proprinter	`print -dibmpro`	Yes
PostScript black and white	`print -dps`	No
PostScript color	`print -dpsc`	No
PostScript Level 2 black and white	`print -dps2`	No
PostScript Level 2 color	`print -dpsc2`	No
Windows color (Windows only)	`print -dwinc`	No
Windows monochrome (Windows only)	`print -dwin`	No

Note Generally, Level 2 PostScript files are smaller and render more quickly when printing than Level 1 PostScript files. However, not all PostScript printers support Level 2, so determine the capabilities of your printer before using those drivers. Level 2 PostScript is the default for UNIX. You can change this default by editing the `printopt.m` file.

Graphics Format Files

To save your figure as a graphics-format file, specify a format switch and filename. To set the resolution of the output file for a built-in MATLAB format, use the `-r` switch. (For example, `-r300` sets the output resolution to 300 dots per inch.) The `-r` switch is also supported for Windows Enhanced Metafiles but is not supported for Ghostscript formats.

The table below shows the supported output formats for exporting from MATLAB and the switch settings to use. In some cases, a format is available both as a MATLAB output filter and as a Ghostscript output filter. The first column indicates this by showing "MATLAB" or "Ghostscript" in parentheses. All formats are supported on both the PC and UNIX platforms.

File Format	Option String (Command line only)
BMP (Ghostscript) Monochrome BMP	`-dbmpmono`
BMP (Ghostscript) 24-bit BMP	`-dbmp16m`
BMP (Ghostscript) 8-bit (256-color) BMP *this format uses a fixed colormap	`-dbmp256`
BMP (MATLAB) 24-bit	`-dbmp`
EMF (MATLAB)	`-dmeta`
EPS (MATLAB) black and white	`-deps`
EPS (MATLAB) color	`-depsc`
EPS (MATLAB) Level 2 black and white	`-deps2`
EPS (MATLAB) Level 2 color	`-depsc2`
HDF (MATLAB) 24-bit	`-dhdf`
ILL (Adobe Illustrator) (MATLAB)	`-dill`
JPEG (MATLAB) 24-bit	`-djpeg`
PBM (Ghostscript) (plain format) 1-bit	`-dpbm`
PBM (Ghostscript) (raw format) 1-bit	`-dpbmraw`
PCX (Ghostscript) 1-bit	`-dpcxmono`
PCX (Ghostscript) 24-bit color PCX file format, three 8-bit planes	`-dpcx24b`
PCX (Ghostscript) 8-bit Newer color PCX file format (256-color)	`-dpcx256`
PCX (Ghostscript) Older color PCX file format (EGA/VGA, 16-color)	`-dpcx16`

File Format	Option String (Command line only)
PCX (MATLAB) 8-bit	`-dpcx`
PDF (Ghostscript) Color PDF file Format	`-dpdf`
PGM (Ghostscript) Portable Graymap (plain format)	`-dpgm`
PGM (Ghostscript) Portable Graymap (raw format)	`-dpgmraw`
PNG (MATLAB) 24-bit	`-dpng`
PPM (Ghostscript) Portable Pixmap, plain format	`-dppm`
PPM (Ghostscript) Portable Pixmap raw format	`-dppmraw`
TIFF (MATLAB) 24-bit	`-dtiff` or `-dtiffn`
TIFF preview for EPS Files	`-tiff`

The TIFF image format is supported on all platforms by almost all word processors for importing images. JPEG is a lossy, highly compressed format that is supported on all platforms for image processing and for inclusion into HTML documents on the World Wide Web. To create these formats, MATLAB renders the figure using the Z-buffer rendering method and the resulting bitmap is then saved to the specified file.

Options

This table summarizes options that you can specify for `print`. The second column also shows which tutorial sections contain more detailed information. The sections listed are located under *Printing and Exporting Figures with MATLAB*.

Option	Description
`-adobecset`	PostScript only. Use PostScript default character set encoding. See Early PostScript 1 Printers.
`-append`	PostScript only. Append figure to existing PostScript file. See Appending Figures to a PostScript File.

Option	Description
`-cmyk`	PostScript only. Print with CMYK colors instead of RGB. See Creating CMYK Output.
`-device`	Printer driver to use. See Specifying a Printer Driver.
`-dsetup`	Display the **Print Setup** dialog.
`-f`*`handle`*	Handle of figure to print. Note that you cannot specify both this option and the `-s`*`windowtitle`* option. See Which Figure Is Printed.
`-loose`	PostScript and Ghostscript only. Use loose bounding box for PostScript. See Producing Uncropped Output.
`-noui`	Suppress printing of user interface controls. See Excluding User Interface Controls from Output.
`-OpenGL`	Render using the OpenGL algorithm. Note that you cannot specify this method in conjunction with `-zbuffer` or `-painters`. See Setting the Rendering Method.
`-painters`	Render using the Painter's algorithm. Note that you cannot specify this method in conjunction with `-zbuffer` or `-OpenGL`. See Setting the Rendering Method.
`-P`*`printer`*	UNIX only. Specify name of printer to use. See Specifying a Printer.
`-r`*`number`*	PostScript and Ghostscript only. Specify resolution in dots per inch. See Setting Resolution.
`-s`*`windowtitle`*	Specify name of Simulink system window to print. Note that you cannot specify both this option and the `-f`*`handle`* option. See Which Figure Is Printed.
`-v`	Windows only. Display the Windows **Print** dialog box. The `v` stands for "verbose mode."
`-zbuffer`	Render using the Z-buffer algorithm. Note that you cannot specify this method in conjunction with `-OpenGL` or `-painters`. See Setting the Rendering Method.

Paper Sizes

MATLAB supports a number of standard paper sizes. You can select from the following list by setting the PaperType property of the figure or selecting a supported paper size from the **Print** dialog box.

Property Value	Size (Width-by-Height)
usletter	8.5-by-11 inches
uslegal	11-by-14 inches
tabloid	11-by-17 inches
A0	841-by-1189mm
A1	594-by-841mm
A2	420-by-594mm
A3	297-by-420mm
A4	210-by-297mm
A5	148-by-210mm
B0	1029-by-1456mm
B1	728-by-1028mm
B2	514-by-728mm
B3	364-by-514mm
B4	257-by-364mm
B5	182-by-257mm
arch-A	9-by-12 inches
arch-B	12-by-18 inches
arch-C	18-by-24 inches
arch-D	24-by-36 inches
arch-E	36-by-48 inches

Property Value	Size (Width-by-Height)
A	8.5-by-11 inches
B	11-by-17 inches
C	17-by-22 inches
D	22-by-34 inches
E	34-by-43 inches

Printing Tips

This section includes information about specific printing issues.

Figures with Resize Functions

The `print` command produces a warning when you print a figure having a callback routine defined for the figure `ResizeFcn`. To avoid the warning, set the figure `PaperPositionMode` property to `auto` or select **Match Figure Screen Size** in the **File->Page Setup...** dialog box.

Troubleshooting MS-Windows Printing

If you encounter problems such as segmentation violations, general protection faults, application errors, or the output does not appear as you expect when using MS-Windows printer drivers, try the following:

- If your printer is PostScript compatible, print with one of MATLAB's built-in PostScript drivers. There are various PostScript device options that you can use with the `print` command: they all start with `–dps`.
- The behavior you are experiencing may occur only with certain versions of the print driver. Contact the print driver vendor for information on how to obtain and install a different driver. If you are using Windows 95, try installing the drivers that ship with the Windows 95 CD-ROM.
- Try printing with one of MATLAB's built-in Ghostscript devices. These devices use Ghostscript to convert PostScript files into other formats, such as HP LaserJet, PCX, Canon BubbleJet, and so on.
- Copy the figure as a Windows Enhanced Metafile using the **Edit-->Copy Figure** menu item on the figure window menu or the `print –dmeta` option at

the command line. You can then import the file into another application for printing.

You can set copy options in the figure's **File-->Preferences...-->Copying Options** dialog box. The Windows Enhanced Metafile clipboard format produces a better quality image than Windows Bitmap.

Printing Thick Lines on Windows95

Due to a limitation in Windows95, MATLAB is set up to print lines as either:

- Solid lines of the specified thickness (`LineWidth`)
- Thin (one pixel wide) lines with the specified line style (`LineStyle`)

If you create lines that are thicker than one pixel and use nonsolid line styles, MATLAB prints these lines with the specified line style, but one pixel wide (i.e., as thin lines).

However, you can change this behavior so that MATLAB prints thick, styled lines as thick, solid lines by editing your `matlab.ini` file, which is in your Windows directory. In this file, find the section,

```
[Matlab Settings]
```

and in this section change the assignment,

```
ThinLineStyles=1
```

to

```
ThinLineStyles=0
```

then restart MATLAB.

Printing MATLAB GUIs

You can generally obtain better results when printing a figure window that contains MATLAB `uicontrols` by setting these key properties:

- Set the figure `PaperPositionMode` property to `auto`. This ensures the printed version is the same size as the onscreen version. With `PaperPositionMode` set to `auto` MATLAB does not resize the figure to fit the current value of the `PaperPosition`. This is particularly important if you have specified a figure

`ResizeFcn` because if MATLAB resizes the figure during the print operation, the `ResizeFcn` is automatically called.

To set `PaperPositionMode` on the current figure, use the command:

```
set(gcf,'PaperPositionMode','auto')
```

- Set the figure `InvertHardcopy` property to `off`. By default, MATLAB changes the figure background color of printed output to white, but does not change the color of uicontrols. If you have set the background color to, for example, match the gray of the GUI devices, you must set `InvertHardcopy` to `off` to preserve the color scheme.

  To set `InvertHardcopy` on the current figure, use the command:

  ```
  set(gcf,'InvertHardcopy','off')
  ```

- Use a color device if you want lines and text that are in color on the screen to be written to the output file as colored objects. Black and white devices convert colored lines and text to black or white to provide the best contrast with the background and to avoid dithering.
- Use the `print` command's `–loose` option to prevent MATLAB from using a bounding box that is tightly wrapped around objects contained in the figure. This is important if you have intentionally used space between uicontrols or axes and the edge of the figure and you want to maintain this appearance in the printed output.

Notes on Printing Interpolated Shading with PostScript Drivers

MATLAB can print surface objects (such as graphs created with `surf` or `mesh`) using interpolated colors. However, only patch objects that are composed of triangular faces can be printed using interpolated shading.

Printed output is always interpolated in RGB space, not in the colormap colors. This means, if you are using indexed color and interpolated face coloring, the printed output can look different from what is displayed on screen.

PostScript files generated for interpolated shading contain the color information of the graphics object's vertices and require the printer to perform the interpolation calculations. This can take an excessive amount of time and in some cases, printers may actually "time-out" before finishing the print job. One solution to this problem is to interpolate the data and generate a greater number of faces, which can then be flat shaded.

To ensure that the printed output matches what you see on the screen, print using the `-zbuffer` option. To obtain higher resolution (for example, to make text look better), use the `–r` option to increase the resolution. There is, however, a trade-off between the resolution and the size of the created PostScript file, which can be quite large at higher resolutions. The default resolution of 150 dpi generally produces good results. You can reduce the size of the output file by making the figure smaller before printing it and setting the figure `PaperPositionMode` to `auto`, or by just setting the `PaperPosition` property to a smaller size.

Note that in some UNIX environments, the default `lpr` command cannot print files larger than 1 Mbyte unless you use the `–s` option, which MATLAB does by default. See the `lpr` man page for more information.

Examples

Specifying the Figure to Print

You can print a noncurrent figure by specifying the figure's handle. If a figure has the title "Figure No. 2", its handle is 2. The syntax is,

```
print -fhandle
```

This example prints the figure whose handle is 2, regardless of which figure is the current figure.

```
print -f2
```

Note Note that you must use the `-f` option if the figure's handle is hidden (i.e., its `HandleVisibility` property is set to `off`).

This example saves the figure with the handle `-f2` to a PostScript file named `Figure2`, which can be printed later.

```
print -f2 -dps 'Figure2.ps'
```

If the figure uses noninteger handles, use the `figure` command to get its value, and then pass it in as the first argument.

```
h = figure('IntegerHandle','off')
```

```
print h -depson
```

You can also pass a figure handle as a variable to the function form of `print`. For example,

```
h = figure; plot(1:4,5:8)
print(h)
```

This example uses the function form of `print` to enable a filename to be passed in as a variable.

```
filename = 'mydata';
print('-f3', '-dpsc', filename);
```

(Because a filename is specified, the figure will be printed to a file.)

Specifying the Model to Print

To print a noncurrent Simulink model, use the `-s` option with the title of the window. For example, this command prints the Simulink window titled `f14`.

```
print -sf14
```

If the window title includes any spaces, you must call the function form rather than the command form of `print`. For example, this command saves a Simulink window title `Thruster Control`.

```
print('-sThruster Control')
```

To print the current system use:

```
print -s
```

For information about issues specific to printing Simulink windows, see the Simulink documentation.

This example prints a surface plot with interpolated shading. Setting the current figure's (`gcf`) `PaperPositionMode` to `auto` enables you to resize the figure window and print it at the size you see on the screen. See Options and the previous section for information on the `–zbuffer` and `–r200` options.

```
surf(peaks)
shading interp
set(gcf,'PaperPositionMode','auto')
print –dpsc2 –zbuffer –r200
```

Batch Processing

You can use the function form of `print` to pass variables containing file names. For example, this for loop creates a series of graphs and prints each one with a different file name.

```
for k=1:length(fnames)
   surf(Z(:,:,k))
   print('-dtiff','-r200',fnames(k))
end
```

Tiff Preview

The command:

```
print -depsc -tiff -r300 picture1
```

saves the current figure at 300 dpi, in a color Encapsulated PostScript file named `picture1.eps`. The `-tiff` option creates a 72 dpi TIFF preview, which many word processor applications can display on screen after you import the EPS file. This enables you to view the picture on screen within your word processor and print the document to a PostScript printer using a resolution of 300 dpi.

See Also

`orient`, `figure`

printdlg

Purpose Display print dialog box

Syntax

```
printdlg
printdlg(fig)
printdlg('-crossplatform',fig)
printdlg('-setup',fig)
```

Description

`printdlg` prints the current figure.

`printdlg(fig)` creates a dialog box from which you can print the figure window identified by the handle `fig`. Note that uimenus do not print.

`printdlg('-crossplatform',fig)` displays the standard cross-platform MATLAB printing dialog rather than the built-in printing dialog box for Microsoft Windows computers. Insert this option before the `fig` argument.

`printdlg('-setup',fig)` forces the printing dialog to appear in a setup mode. Here one can set the default printing options without actually printing.

Purpose

Preview figure to be printed

Syntax

```
printpreview
printpreview(f)
```

Description

`printpreview` displays a dialog box showing the figure in the currently active figure window as it will be printed. The figure is displayed with a `1/4` size thumbnail or full size image.

`printpreview(f)` displays a dialog box showing the figure having the handle `f` as it will be printed.

You can select any of the following options from the **Print Preview** dialog box.

Option Button	Description
`Print...`	Close **Print Preview** and open the **Print** dialog
`Page Setup...`	Open the **Page Setup** dialog
`Zoom In`	Display a full size image of the page
`Zoom Out`	Display a 1/4 scaled image of the page
`Close`	Close the **Print Preview** dialog

See Also

`printdlg`, `pagesetupdlg`

prod

Purpose

Product of array elements

Syntax

```
B = prod(A)
B = prod(A,dim)
```

Description

`B = prod(A)` returns the products along different dimensions of an array.

If `A` is a vector, `prod(A)` returns the product of the elements.

If `A` is a matrix, `prod(A)` treats the columns of `A` as vectors, returning a row vector of the products of each column.

If `A` is a multidimensional array, `prod(A)` treats the values along the first non-singleton dimension as vectors, returning an array of row vectors.

`B = prod(A,dim)` takes the products along the dimension of `A` specified by scalar `dim`.

Examples

The magic square of order 3 is

```
M = magic(3)

M =
    8    1    6
    3    5    7
    4    9    2
```

The product of the elements in each column is

```
prod(M) =

    96    45    84
```

The product of the elements in each row can be obtained by:

```
prod(M,2) =

     48
    105
     72
```

See Also

`cumprod`, `diff`, `sum`

Purpose

Tool for optimizing and debugging M-file code

Syntax

```
profile on
profile on -detail level
profile on -history
profile off
profile resume
profile clear
profile report
profile report basename
profile plot
s = profile('status')
stats = profile('info')
```

Description

The profiler utility helps you debug and optimize M-files by tracking their execution time. For each function in the M-file, the profiler records information about execution time, number of calls, parent functions, child functions, code line hit count, and code line execution time. Some people use `profile` simply to see the child functions; see also `depfun` for that purpose.

`profile` **`on`** starts the profiler, clearing previously recorded profile statistics.

`profile` **`on`** -**`detail`** *`level`* starts the profiler for the set of functions specified by *`level`*, clearing previously recorded profile statistics.

Value for level	Functions Profiler Gathers Information About
`mmex`	M-functions, M-subfunctions, and MEX-functions; `mmex` is the default value
`builtin`	Same functions as for `mmex` plus built-in functions such as `eig`
`operator`	Same functions as for `builtin` plus built-in operators such as `+`

`profile` **`on -history`** starts the profiler, clearing previously recorded profile statistics, and recording the exact sequence of function calls. The profiler records up to 10,000 function entry and exit events. For more than 10,000 events, the profiler continues to record other profile statistics, but not the sequence of calls.

`profile` **`off`** suspends the profiler.

`profile` **`resume`** restarts the profiler without clearing previously recorded statistics.

`profile` **`clear`** clears the statistics recorded by the profiler.

`profile` **`report`** suspends the profiler, generates a profile report in HTML format, and displays the report in your system's default Web browser.

`profile` **`report`** `basename` suspends the profiler, generates a profile report in HTML format, saves the report in the file `basename` in the current directory, and displays the report in your system's default Web browser. Because the report consists of several files, do not provide an extension for `basename`.

`profile` **`plot`** suspends the profiler and displays in a figure window a bar graph of the functions using the most execution time.

`s = profile(`**`'status'`**`)` displays a structure containing the current profiler status. The structure's fields are shown below.

Field	Values
`ProfilerStatus`	`'on'` or `'off'`
`DetailLevel`	`'mmex'`, `'builtin'`, or `'operator'`
`HistoryTracking`	`'on'` or `'off'`

`stats = profile('info')` suspends the profiler and displays a structure containing profiler results. Use this function to access the data generated by the profiler. The structure's fields are

`FunctionTable`	Array containing list of all functions called
`FunctionHistory`	Array containing function call history
`ClockPrecision`	Precision of profiler's time measurement

Remarks

To see an example of a profile report and profile plot, as well as to learn more about the results and how to use profiling, see "Improving M-File Performance: the Profiler" in *Using MATLAB*.

Examples

Follow these steps to run the profiler and create a profile report.

1 Run the profiler for code that computes the Lotka-Volterra predator-prey population model.

```
profile on -detail builtin -history
[t,y] = ode23('lotka',[0 2],[20;20]);
profile report
```

The profile report appears in your system's default Web browser, providing information for all M-functions, M-subfunctions, MEX-functions, and built-in functions. The report includes the function call history.

2 Generate the profile plot.

```
profile plot
```

The profile plot appears in a figure window.

3 Because the report and plot features suspend the profiler, resume its operation without clearing the statistics already gathered.

```
profile resume
```

The profiler will continue gathering statistics when you execute the next M-file.

See Also

`depdir`, `depfun`, `profreport`, `tic`

"Improving M-File Performance – the Profiler" in *Using MATLAB*

profreport

Purpose

Generate profile report

Syntax

```
profreport
profreport(basename)
profreport(stats)
profreport(basename,stats)
```

Description

`profreport` suspends the profiler, generates a profile report in HTML format using the current profiler results, and displays the report in a Web browser.

`profreport(basename)` suspends the profiler, generates a profile report in HTML format using the current profiler results, saves the report using the `basename` you supply, and displays the report in a Web browser. Because the report consists of several files, do not provide an extension for `basename`.

`profreport(stats)` suspends the profiler, generates a profile report in HTML format using the profiler results `info`, and displays the report in a Web browser. `stats` is the profiler information structure returned by `stats = profile('info')`.

`profreport(basename,stats)` suspends the profiler, generates a profile report in HTML format using the profiler results `stats`, saves the report using the `basename` you supply, and displays the report in a Web browser. `stats` is the profiler information structure returned by `stats = profile('info')`. Because the report consists of several files, do not provide an extension for `basename`.

Examples

Run profiler and view the structure containing profile results.

1 Run the profiler for code that computes the Lotka-Volterra predator-prey population model.

   ```
   profile on -detail builtin -history
   [t,y] = ode23('lotka',[0 2],[20;20]);
   ```

2 View the structure containing the profile results.

```
stats = profile('info')
```

MATLAB returns

```
stats =
FunctionTable: [42x1 struct]
    FunctionHistory: [2x830 double]
     ClockPrecision: 0.0100
               Name: 'MATLAB'
```

3 View the contents of the second element in the `FunctionTable` structure.

```
stats.FunctionTable(2)
```

MATLAB returns

```
ans =
          FunctionName: 'horzcat'
              FileName: ''
                  Type: 'Builtin-function'
              NumCalls: 43
             TotalTime: 0
    TotalRecursiveTime: 0
              Children: [0x1 struct]
               Parents: [2x1 struct]
         ExecutedLines: [0x3 double]
```

4 Display the profile report from the structure.

```
profreport(stats)
```

MATLAB displays the profile report in a Web browser.

See Also

`profile`

"Improving M-File Performance: the Profiler" in Using MATLAB

propedit

Purpose Starts the Property Editor

Syntax

```
propedit
propedit(HandleList)
```

Description `propedit` starts the Property Editor, a graphical user interface to the properties of Handle Graphics objects. If you call it without any input arguments, the Property Editor displays the properties of the current figure, if there are more than one figure displayed, or the root object, if there is no currently active figure.

`propedit(HandleList)` edits the properties for the object (or objects) in `HandleList`.

Note Starting the Property Editor enables plot editing mode for the figure.

---

propedit (activex)

Purpose
Request the control to display its built-in property page.

Syntax

```
propedit (a)
```

Arguments

`a`
An interface handle previously returned from `actxcontrol`, `get`, or `invoke`.

Description
Request the control to display its built-in property page. Note that some controls do not have a built-in property page. For those objects, this command will fail.

Example

```
propedit (a)
```

pwd

Purpose

Display current directory

Graphical Interface

As an alternative to the `pwd` function, use the Current Directory field in the MATLAB desktop toolbar.

Syntax

```
pwd
s = pwd
```

Description

`pwd` displays the current working directory.

`s = pwd` returns the current directory to the variable `s`.

See Also

`cd`, `dir`, `path`, `what`

Purpose Quasi-Minimal Residual method

Syntax

```
x = qmr(A,b)
qmr(A,b,tol)
qmr(A,b,tol,maxit)
qmr(A,b,tol,maxit,M)
qmr(A,b,tol,maxit,M1,M2)
qmr(A,b,tol,maxit,M1,M2,x0)
qmr(afun,b,tol,maxit,m1fun,m2fun,x0,p1,p2,...)
[x,flag] = qmr(A,b,...)
[x,flag,relres] = qmr(A,b,...)
[x,flag,relres,iter] = qmr(A,b,...)
[x,flag,relres,iter,resvec] = qmr(A,b,...)
```

Description

`x = qmr(A,b)` attempts to solve the system of linear equations `A*x=b` for `x`. The `n`-by-`n` coefficient matrix `A` must be square and the column vector `b` must have length `n`. `A` can be a function `afun` such that `afun(x)` returns `A*x` and `afun(x,'transp')` returns `A'*x`.

If `qmr` converges, a message to that effect is displayed. If `qmr` fails to converge after the maximum number of iterations or halts for any reason, a warning message is printed displaying the relative residual `norm(b-A*x)/norm(b)` and the iteration number at which the method stopped or failed.

`qmr(A,b,tol)` specifies the tolerance of the method. If `tol` is `[]`, then `qmr` uses the default, `1e-6`.

`qmr(A,b,tol,maxit)` specifies the maximum number of iterations. If `maxit` is `[]`, then `qmr` uses the default, `min(n,20)`.

`qmr(A,b,tol,maxit,M)` and `qmr(A,b,tol,maxit,M1,M2)` use preconditioners `M` or `M = M1*M2` and effectively solve the system `inv(M)*A*x = inv(M)*b` for `x`. If `M` is `[]` then `qmr` applies no preconditioner. `M` can be a function `mfun` such that `mfun(x)` returns `M\x` and `mfun(x,'transp')` returns `M'\x`.

`qmr(A,b,tol,maxit,M1,M2,x0)` specifies the initial guess. If `x0` is `[]`, then `qmr` uses the default, an all zero vector.

`qmr(afun,b,tol,maxit,m1fun,m2fun,x0,p1,p2,...)` passes parameters `p1,p2,...` to functions `afun(x,p1,p2,...)` and `afun(x,p1,p2,...,'transp')` and similarly to the preconditioner functions `m1fun` and `m2fun`.

`[x,flag] = qmr(A,b,...)` also returns a convergence flag.

Flag	Convergence
0	`qmr` converged to the desired tolerance `tol` within `maxit` iterations.
1	`qmr` iterated `maxit` times but did not converge.
2	Preconditioner `M` was ill-conditioned.
3	The method stagnated. (Two consecutive iterates were the same.)
4	One of the scalar quantities calculated during `qmr` became too small or too large to continue computing.

Whenever `flag` is not `0`, the solution `x` returned is that with minimal norm residual computed over all the iterations. No messages are displayed if the `flag` output is specified.

`[x,flag,relres] = qmr(A,b,...)` also returns the relative residual `norm(b-A*x)/norm(b)`. If `flag` is `0`, `relres <= tol`.

`[x,flag,relres,iter] = qmr(A,b,...)` also returns the iteration number at which `x` was computed, where `0 <= iter <= maxit`.

`[x,flag,relres,iter,resvec] = qmr(A,b,...)` also returns a vector of the residual norms at each iteration, including `norm(b-A*x0)`.

Examples

Example 1.

```
n = 100;
on = ones(n,1);
A = spdiags([-2*on 4*on -on],-1:1,n,n);
b = sum(A,2);
```

```
tol = 1e-8; maxit = 15;
M1 = spdiags([on/(-2) on],-1:0,n,n);
M2 = spdiags([4*on -on],0:1,n,n);
x = qmr(A,b,tol,maxit,M1,M2,[]);
```

Alternatively, use this matrix-vector product function

```
function y = afun(x,n,transp_flag)
if (nargin > 2) & strcmp(transp_flag,'transp')
   y = 4 * x;
   y(1:n-1) = y(1:n-1) - 2 * x(2:n);
   y(2:n) = y(2:n) - x(1:n-1);
else
   y = 4 * x;
   y(2:n) = y(2:n) - 2 * x(1:n-1);
   y(1:n-1) = y(1:n-1) - x(2:n);
end
```

as input to `qmr`

```
x1 = qmr(@afun,b,tol,maxit,M1,M2,[],n);
```

Example 2.

```
load west0479;
A = west0479;
b = sum(A,2);
[x,flag] = qmr(A,b)
```

`flag` is `1` because `qmr` does not converge to the default tolerance `1e-6` within the default 20 iterations.

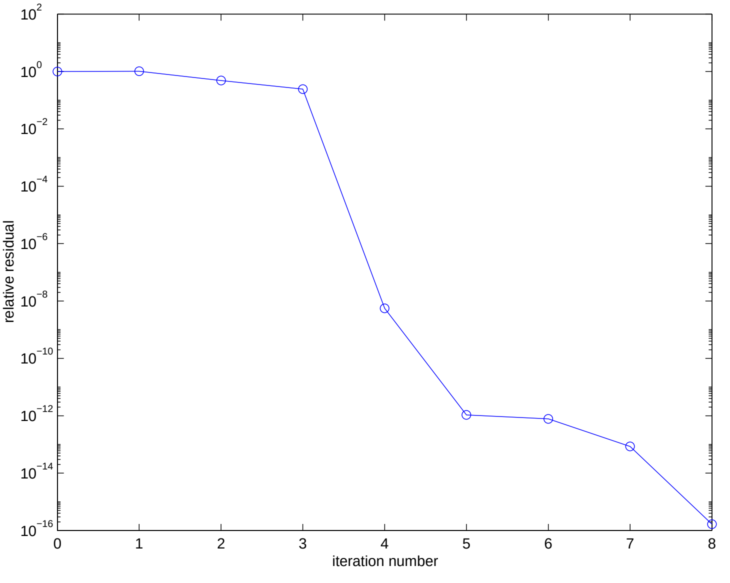
```
[L1,U1] = luinc(A,1e-5);
[x1,flag1] = qmr(A,b,1e-6,20,L1,U1)
```

`flag1` is `2` because the upper triangular `U1` has a zero on its diagonal, and `qmr` fails in the first iteration when it tries to solve a system such as `U1*y = r` for `y` using backslash.

```
[L2,U2] = luinc(A,1e-6);
[x2,flag2,relres2,iter2,resvec2] = qmr(A,b,1e-15,10,L2,U2)
```

`flag2` is `0` because `qmr` converges to the tolerance of `1.6571e-016` (the value of `relres2`) at the eighth iteration (the value of `iter2`) when preconditioned by the incomplete LU factorization with a drop tolerance of `1e-6`. `resvec2(1) = norm(b)` and `resvec2(9) = norm(b-A*x2)`. You can follow the progress of `qmr` by plotting the relative residuals at each iteration starting from the initial estimate (iterate number `0`).

```
semilogy(0:iter2,resvec2/norm(b),'-o')
xlabel('iteration number')
ylabel('relative residual')
```



See Also

`bicg`, `bicgstab`, `cgs`, `gmres`, `lsqr`, `luinc`, `minres`, `pcg`, `symmlq`

`@` (function handle), `\` (backslash)

References

[1] Barrett, R., M. Berry, T. F. Chan, et al., *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*, SIAM, Philadelphia, 1994.

[2] Freund, Roland W. and Nöel M. Nachtigal, “QMR: A quasi-minimal residual method for non-Hermitian linear systems”, *SIAM Journal: Numer. Math.* 60, 1991, pp. 315-339.

Purpose Orthogonal-triangular decomposition

Syntax

```
[Q,R] = qr(A)        (full and sparse matrices)
[Q,R] = qr(A,0)      (full and sparse matrices)
[Q,R,E] = qr(A)      (full matrices)
[Q,R,E] = qr(A,0)    (full matrices)
X = qr(A)            (full matrices)
R = qr(A)            (sparse matrices)
[C,R] = qr(A,B)      (sparse matrices)
R = qr(A,0)          (sparse matrices)
[C,R] = qr(A,B,0)    (sparse matrices)
```

Description The `qr` function performs the orthogonal-triangular decomposition of a matrix. This factorization is useful for both square and rectangular matrices. It expresses the matrix as the product of a real orthonormal or complex unitary matrix and an upper triangular matrix.

`[Q,R] = qr(A)` produces an upper triangular matrix `R` of the same dimension as `A` and a unitary matrix `Q` so that `A = Q*R`. For sparse matrices, `Q` is often nearly full. If `[m n] = size(A)`, then `Q` is `m`-by-`m` and `R` is `m`-by-`n`.

`[Q,R] = qr(A,0)` produces an "economy-size" decomposition. If `[m n] = size(A)`, and `m > n`, then `qr` computes only the first `n` columns of of `Q` and `R` is `n`-by-`n`.

`[Q,R,E] = qr(A)` for full matrix `A`, produces a permutation matrix `E`, an upper triangular matrix `R` with decreasing diagonal elements, and a unitary matrix `Q` so that `A*E = Q*R`. The column permutation `E` is chosen so that `abs(diag(R))` is decreasing.

`[Q,R,E] = qr(A,0)` for full matrix `A`, produces an "economy-size" decomposition in which `E` is a permutation vector, so that `Q*R = A(:,E)`. The column permutation `E` is chosen so that `abs(diag(R))` is decreasing.

`X = qr(A)` for full matrix `A`, returns the output of the LAPACK subroutine `DGEQRF` or `ZGEQRF`. `triu(qr(A))` is `R`.

`R = qr(A)` for sparse matrix `A`, produces only an upper triangular matrix, `R`. The matrix `R` provides a Cholesky factorization for the matrix associated with the normal equations,

```
R'*R = A'*A
```

This approach avoids the loss of numerical information inherent in the computation of `A'*A`.

`[C,R] = qr(A,B)` for sparse matrix `A`, applies the orthogonal transformations to `B`, producing `C = Q'*B` without computing `Q`. `B` and `A` must have the same number of rows.

`R = qr(A,0)` and `[C,R] = qr(A,B,0)` for sparse matrix `A`, produce "economy-size" results.

For sparse matrices, the Q-less QR factorization allows the solution of sparse least squares problems

$$\text{minimize}\|Ax - b\|$$

with two steps

```
[C,R] = qr(A,b)
x = R\c
```

If `A` is sparse but not square, MATLAB uses the two steps above for the linear equation solving backslash operator, i.e., `x = A\b`.

Examples

Example 1. Start with

```
A =  [ 1      2      3
       4      5      6
       7      8      9
      10     11     12 ]
```

This is a rank-deficient matrix; the middle column is the average of the other two columns. The rank deficiency is revealed by the factorization:

```
[Q,R] = qr(A)

Q =

    -0.0776    -0.8331     0.5444     0.0605
```

```
    -0.3105    -0.4512    -0.7709     0.3251
    -0.5433    -0.0694    -0.0913    -0.8317
    -0.7762     0.3124     0.3178     0.4461

R =

    -12.8841    -14.5916    -16.2992
           0     -1.0413     -2.0826
           0           0      0.0000
           0           0           0
```

The triangular structure of R gives it zeros below the diagonal; the zero on the diagonal in R(3,3) implies that R, and consequently A, does not have full rank.

Example 2. This examples uses matrix A from the first example. The QR factorization is used to solve linear systems with more equations than unknowns. For example, let

```
b = [1;3;5;7]
```

The linear system $Ax = b$ represents four equations in only three unknowns. The best solution in a least squares sense is computed by

```
x = A\b
```

which produces

```
Warning: Rank deficient, rank = 2, tol = 1.4594E-014
x =
    0.5000
         0
    0.1667
```

The quantity tol is a tolerance used to decide if a diagonal element of R is negligible. If [Q,R,E] = qr(A), then

```
tol = max(size(A))*eps*abs(R(1,1))
```

The solution x was computed using the factorization and the two steps

```
y = Q'*b;
x = R\y
```

The computed solution can be checked by forming Ax. This equals b to within roundoff error, which indicates that even though the simultaneous equations $Ax = b$ are overdetermined and rank deficient, they happen to be consistent. There are infinitely many solution vectors x; the QR factorization has found just one of them.

Algorithm

The qr function uses LAPACK routines to compute the QR decomposition:

Syntax	Real	Complex
`R = qr(A)` `R = qr(A,0)`	DGEQRF	ZGEQRF
`[Q,R] = qr(A)` `[Q,R] = qr(A,0)`	DGEQRF, DORGQR	ZGEQRF, ZUNGQR
`[Q,R,e] = qr(A)` `[Q,R,e] = qr(A,0)`	DGEQPF, DORGQR	ZGEQPF, ZUNGQR

See Also

lu, null, orth, qrdelete, qrinsert, qrupdate

The arithmetic operators \ and /

References

[1] Anderson, E., Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, *LAPACK User's Guide* (http://www.netlib.org/lapack/lug/lapack_lug.html), Third Edition, SIAM, Philadelphia, 1999.

Purpose

Delete column from QR factorization

Syntax

```
[Q,R] = qrdelete(Q,R,j)
```

Description

`[Q,R] = qrdelete(Q,R,j)` changes `Q` and `R` to be the factorization of the matrix `A` with its jth column, `A(:,j)`, removed.

Inputs `Q` and `R` represent the original QR factorization of matrix `A`, as returned by the statement `[Q,R] = qr(A)`. Argument `j` specifies the column to be removed from matrix `A`.

Algorithm

The `qrdelete` function uses a series of Givens rotations to zero out the appropriate elements of the factorization.

See Also

`qr`, `qrinsert`

qrinsert

Purpose Insert column in QR factorization

Syntax

```
[Q,R] = qrinsert(Q,R,j,x)
```

Description `[Q,R] = qrinsert(Q,R,j,x)` changes `Q` and `R` to be the factorization of the matrix obtained by inserting an extra column, `x`, before `A(:,j)`. If `A` has `n` columns and `j = n+1`, then `qrinsert` inserts `x` after the last column of `A`.

Inputs `Q` and `R` represent the original QR factorization of matrix `A`, as returned by the statement `[Q,R] = qr(A)`. Argument `x` is the column vector to be inserted into matrix `A`. Argument `j` specifies the column before which `x` is inserted.

Algorithm The `qrinsert` function inserts the values of `x` into the jth column of `R`. It then uses a series of Givens rotations to zero out the nonzero elements of `R` on and below the diagonal in the jth column.

See Also `qr`, `qrdelete`

Description Rank 1 update to QR factorization

Syntax

```
[Q1,R1] = qrupdate(Q,R,u,v)
```

Description `[Q1,R1] = qrupdate(Q,R,u,v)` when `[Q,R] = qr(A)` is the original QR factorization of `A`, returns the QR factorization of `A + u*v'`, where `u` and `v` are column vectors of appropriate lengths.

Remarks `qrupdate` works only for full matrices.

Examples The matrix

```
mu = sqrt(eps)

mu =

   1.4901e-08

A = [ones(1,4); mu*eye(4)];
```

is a well-known example in least squares that indicates the dangers of forming `A'*A`. Instead, we work with the QR factorization – orthonormal Q and upper triangular R.

```
[Q,R] = qr(A);
```

As we expect, `R` is upper triangular.

```
R =

   -1.0000   -1.0000   -1.0000   -1.0000
         0    0.0000    0.0000    0.0000
         0         0    0.0000    0.0000
         0         0         0    0.0000
         0         0         0         0
```

In this case, the upper triangular entries of `R`, excluding the first row, are on the order of `sqrt(eps)`.

Consider the update vectors

```
u = [-1 0 0 0 0]'; v = ones(4,1);
```

Instead of computing the rather trivial QR factorization of this rank one update to A from scratch with

```
[QT,RT] = qr(A + u*v')

QT =

     0     0     0     0     1
    -1     0     0     0     0
     0    -1     0     0     0
     0     0    -1     0     0
     0     0     0    -1     0

RT =

   1.0e-007 *

    -0.1490         0         0         0
          0   -0.1490         0         0
          0         0   -0.1490         0
          0         0         0   -0.1490
          0         0         0         0
```

we may use qrupdate.

```
[Q1,R1] = qrupdate(Q,R,u,v)

Q1 =

    -0.0000   -0.0000   -0.0000   -0.0000    1.0000
     1.0000   -0.0000   -0.0000   -0.0000    0.0000
     0.0000    1.0000   -0.0000   -0.0000    0.0000
     0.0000    0.0000    1.0000   -0.0000    0.0000
    -0.0000   -0.0000   -0.0000    1.0000    0.0000

R1 =

    1.0e-007 *
     0.1490    0.0000    0.0000    0.0000
          0    0.1490    0.0000    0.0000
          0         0    0.1490    0.0000
```

```
        0         0         0    0.1490
        0         0         0         0
```

Note that both factorizations are correct, even though they are different.

Algorithm

`qrupdate` uses the algorithm in section 12.5.1 of the third edition of *Matrix Computations* by Golub and van Loan. `qrupdate` is useful since, if we take `N = max(m,n)`, then computing the new QR factorization from scratch is roughly an $O(\mathrm{N}^3)$ algorithm, while simply updating the existing factors in this way is an $O(\mathrm{N}^2)$ algorithm.

References

[1] Golub, Gene H. and Charles Van Loan, *Matrix Computations*, Third Edition, Johns Hopkins University Press, Baltimore, 1996

See Also

`cholupdate`, `qr`

Purpose Numerically evaluate integral, adaptive Simpson quadrature

---

Note The `quad8` function, which implemented a higher order method, is obsolete. The `quadl` function is its recommended replacement.

Syntax

```
q = quad(fun,a,b)
q = quad(fun,a,b,tol)
q = quad(fun,a,b,tol,trace)
q = quad(fun,a,b,tol,trace,p1,p2,...)
[q,fcnt] = quadl(fun,a,b,...)
```

Description *Quadrature* is a numerical method used to find the area under the graph of a function, that is, to compute a definite integral.

$$q = \int_a^b f(x)\,dx$$

`q = quad(fun,a,b)` approximates the integral of function `fun` from `a` to `b` to within an error of 10^{-6} using recursive adaptive Simpson quadrature. `fun` accepts a vector `x` and returns a vector `y`, the function `fun` evaluated at each element of `x`.

`q = quad(fun,a,b,tol)` uses an absolute error tolerance tol instead of the default which is `1.0e-6`. Larger values of `tol` result in fewer function evaluations and faster computation, but less accurate results. In MATLAB version 5.3 and earlier, the `quad` function used a less reliable algorithm and a default relative tolerance of `1.0e-3`.

`q = quad(fun,a,b,tol,trace)` with non-zero `trace` shows the values of `[fcnt a b-a Q]` during the recursion.

`q = quad(fun,a,b,tol,trace,p1,p2,...)` provides for additional arguments `p1,p2,...` to be passed directly to function `fun`, `fun(x,p1,p2,...)`. Pass empty matrices for `tol` or `trace` to use the default values.

`[q,fcnt] = quad(...)` returns the number of function evaluations.

The function `quadl` may be more efficient with high accuracies and smooth integrands.

Examples

You can specify `fun` three different ways:

- A string expression involving a single variable

```
Q = quad('1./(x.^3-2*x-5)',0,2);
```

- An inline object

```
F = inline('1./(x.^3-2*x-5)');
Q = quad(F,0,2);
```

- A function handle

```
Q = quad(@myfun,0,2);
```

  where `myfun.m` is an M-file.

```
function y = myfun(x)
y = 1./(x.^3-2*x-5);
```

Algorithm

`quad` implements a low order method using an adaptive recursive Simpson's rule.

Diagnostics

`quad` may issue one of the following warnings:

`'Minimum step size reached'` indicates that the recursive interval subdivision has produced a subinterval whose length is on the order of roundoff error in the length of the original interval. A nonintegrable singularity is possible.

`'Maximum function count exceeded'` indicates that the integrand has been evaluated more than 10,000 times. A nonintegrable singularity is likely.

`'Infinite or Not-a-Number function value encountered'` indicates a floating point overflow or division by zero during the evaluation of the integrand in the interior of the interval.

See Also

`dblquad`, `inline`, `quadl`, `@` (function handle)

References

[1] Gander, W. and W. Gautschi, “Adaptive Quadrature – Revisited”, BIT, Vol. 40, 2000, pp. 84-101. This document is also available at `http://www.inf.ethz.ch/personal/gander`.

Purpose

Numerically evaluate integral, adaptive Lobatto quadrature

Syntax

```
q = quadl(fun,a,b)
q = quadl(fun,a,b,tol)
q = quadl(fun,a,b,tol,trace)
q = quadl(fun,a,b,tol,trace,p1,p2,...)
[q,fcnt] = quadl(fun,a,b,...)
```

Description

`q = quadl(fun,a,b)` approximates the integral of function `fun` from `a` to `b`, to within an error of 10^{-6} using recursive adaptive Lobatto quadrature. `fun` accepts a vector `x` and returns a vector `y`, the function `fun` evaluated at each element of `x`.

`q = quadl(fun,a,b,tol)` uses an absolute error tolerance of `tol` instead of the default, which is `1.0e-6`. Larger values of `tol` result in fewer function evaluations and faster computation, but less accurate results.

`quadl(fun,a,b,tol,trace)` with non-zero `trace` shows the values of `[fcnt a b-a q]` during the recursion.

`quadl(fun,a,b,tol,trace,p1,p2,...)` provides for additional arguments `p1,p2,...` to be passed directly to function `fun`, `fun(x,p1,p2,...)`. Pass empty matrices for `tol` or `trace` to use the default values.

`[q,fcnt] = quadl(...)` returns the number of function evaluations.

Use array operators `.*`, `./` and `.^` in the definition of `fun` so that it can be evaluated with a vector argument.

The function `quad` may be more efficient with low accuracies or nonsmooth integrands.

Examples

You can specify `fun` three different ways:

- A string expression involving a single variable

```
Q = quadl('1./(x.^3-2*x-5)',0,2);
```

- An inline object

```
F = inline('1./(x.^3-2*x-5)');
Q = quadl(F,0,2);
```

- A function handle

```
Q = quadl(@myfun,0,2);
```

where `myfun.m` is an M-file.

```
function y = myfun(x)
y = 1./(x.^3-2*x-5);
```

Algorithm

`quadl` implements a high order method using an adaptive Gauss/Lobatto qudrature rule.

Diagnostics

`quadl` may issue one of the following warnings:

`'Minimum step size reached'` indicates that the recursive interval subdivision has produced a subinterval whose length is on the order of roundoff error in the length of the original interval. A nonintegrable singularity is possible.

`'Maximum function count exceeded'` indicates that the integrand has been evaluated more than 10,000 times. A nonintegrable singularity is likely.

`'Infinite or Not-a-Number function value encountered'` indicates a floating point overflow or division by zero during the evaluation of the integrand in the interior of the interval.

See Also

`dblquad`, `inline`, `quad`, `@` (function handle)

References

[1] Gander, W. and W. Gautschi, “Adaptive Quadrature – Revisited”, BIT, Vol. 40, 2000, pp. 84-101. This document is also available at `http://www.inf.ethz.ch/personal/gander`.

Purpose

Create and display question dialog box

Syntax

```
button = questdlg('qstring')
button = questdlg('qstring','title')
button = questdlg('qstring','title','default')
button = questdlg('qstring','title','str1','str2','default')
button =
  questdlg('qstring','title','str1','str2','str3','default')
```

Description

`button = questdlg('qstring')` displays a modal dialog presenting the question `'qstring'`. The dialog has three default buttons, **Yes**, **No**, and **Cancel**. `'qstring'` is a cell array or a string that automatically wraps to fit within the dialog box. `button` contains the name of the button pressed.

`button = questdlg('qstring','title')` displays a question dialog with `'`*`title`*`'` displayed in the dialog's title bar.

`button = questdlg('qstring','title','default')` specifies which push button is the default in the event that the **Return** key is pressed. `'`*`default`*`'` must be `'Yes'`, `'No'`, or `'Cancel'`.

`button = questdlg('qstring','title','str1','str2','default')` creates a question dialog box with two push buttons labeled `'str1'` and `'str2'`. `'default'` specifies the default button selection and must be `'str1'` or `'str2'`.

`button = questdlg('qstring','title','str1','str2','str3','default')` creates a question dialog box with three push buttons labeled `'str1'`, `'str2'`, and `'str3'`. `'default'` specifies the default button selection and must be `'str1'`, `'str2'`, or `'str3'`.

Example

Create a question dialog asking the user whether to continue a hypothetical operation:

```
button = questdlg('Do you want to continue?',...
'Continue Operation','Yes','No','Help','No');
if strcmp(button,'Yes')
   disp('Creating file')
```

```
elseif strcmp(button,'No')
   disp('Canceled file operation')
elseif strcmp(button,'Help')
   disp('Sorry, no help available')
end
```

See Also

dialog, errordlg, helpdlg, inputdlg, msgbox, warndlg

Purpose Terminate MATLAB

Graphical Interface As an alternative to the `quit` function, use the close box or select **Exit MATLAB** from the **File** menu in the MATLAB desktop.

Syntax

```
quit
quit cancel
quit force
```

Description `quit` terminates MATLAB after running `finish.m`, if `finish.m` exists. The workspace is not automatically saved by `quit`. To save the workspace or perform other actions when quitting, create a `finish.m` file to perform those actions. If an error occurs while `finish.m` is running, `quit` is canceled so that you can correct your `finish.m` file without losing your workspace.

`quit` **`cancel`** is for use in `finish.m` and cancels quitting. It has no effect anywhere else.

`quit` **`force`** bypasses `finish.m` and terminates MATLAB. Use this to override `finish.m`, for example, if an errant `finish.m` will not let you quit.

Remarks When using Handle Graphics in `finish.m`, use `uiwait`, `waitfor`, or `drawnow` so that figures are visible. See the reference pages for these functions for more information.

Examples

Two sample `finish.m` files are included with MATLAB. Use them to help you create your own `finish.m`, or rename one of the files to `finish.m` to use it.

- `finishsav.m` – saves the workspace to a MAT-file when MATLAB quits
- `finishdlg.m` – displays a dialog allowing you to cancel quitting; it uses `quit cancel` and contains the following code.

```
button = questdlg('Ready to quit?', ...
                            'Exit Dialog','Yes','No','No');
switch button
            case 'Yes',
              disp('Exiting MATLAB');
              %Save variables to matlab.mat
              save
            case 'No',
              quit cancel;
end
```

See Also

`finish`, `save`, `startup`

Purpose Quiver or velocity plot

Syntax

```
quiver(U,V)
quiver(X,Y,U,V)
quiver(...,scale)
quiver(...,LineSpec)
quiver(...,LineSpec,'filled')
h = quiver(...)
```

Description A quiver plot displays vectors with components (u,v) at the points (x,y).

`quiver(U,V)` draws vectors specified by `U` and `V` at the coordinates defined by `x = 1:n` and `y = 1:m`, where `[m,n] = size(U) = size(V)`. This syntax plots `U` and `V` over a geometrically rectangular grid. `quiver` automatically scales the vectors based on the distance between them to prevent them from overlapping.

`quiver(X,Y,U,V)` draws vectors at each pair of elements in `X` and `Y`. If `X` and `Y` are vectors, `length(X) = n` and `length(Y) = m`, where `[m,n] = size(U) = size(V)`. The vector `X` corresponds to the columns of `U` and `V`, and vector `Y` corresponds to the rows of `U` and `V`.

`quiver(...,scale)` automatically scales the vectors to prevent them from overlapping, then multiplies them by `scale`. `scale = 2` doubles their relative length and `scale = 0.5` halves them. Use `scale = 0` to plot the velocity vectors without the automatic scaling.

`quiver(...,LineSpec)` specifies line style, marker symbol, and color using any valid `LineSpec`. `quiver` draws the markers at the origin of the vectors.

`quiver(...,LineSpec,'filled')` fills markers specified by `LineSpec`.

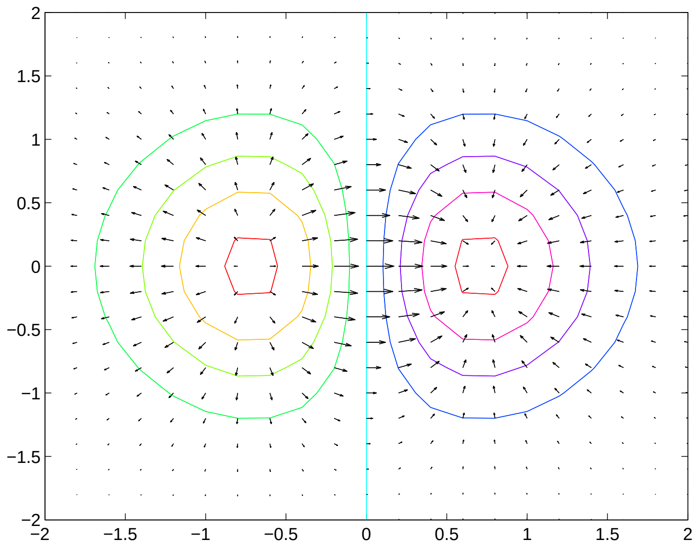
`h = quiver(...)` returns a vector of line handles.

Remarks If `X` and `Y` are vectors, this function behaves as

```
[X,Y] = meshgrid(x,y)
quiver(X,Y,U,V)
```

Examples Plot the gradient field of the function $z = xe^{(-x^2 - y^2)}$.

```
[X,Y] = meshgrid(-2:.2:2);
Z = X.*exp(-X.^2 - Y.^2);
[DX,DY] = gradient(Z,.2,.2);
contour(X,Y,Z)
hold on
quiver(X,Y,DX,DY)
colormap hsv
grid off
hold off
```



See Also `contour`, `LineSpec`, `plot`, `quiver3`

Purpose Three-dimensional velocity plot

Syntax

```
quiver3(Z,U,V,W)
quiver3(X,Y,Z,U,V,W)
quiver3(...,scale)
quiver3(...,LineSpec)
quiver3(...,LineSpec,'filled')
h = quiver3(...)
```

Description A three-dimensional quiver plot displays vectors with components (u,v,w) at the points (x,y,z).

`quiver3(Z,U,V,W)` plots the vectors at the equally spaced surface points specified by matrix `Z`. `quiver3` automatically scales the vectors based on the distance between them to prevent them from overlapping.

`quiver3(X,Y,Z,U,V,W)` plots vectors with components (u,v,w) at the points (x,y,z). The matrices `X`, `Y`, `Z`, `U`, `V`, `W` must all be the same size and contain the corresponding position and vector components.

`quiver3(...,scale)` automatically scales the vectors to prevent them from overlapping, then multiplies them by `scale`. `scale = 2` doubles their relative length and `scale = 0.5` halves them. Use `scale = 0` to plot the vectors without the automatic scaling.

`quiver3(...,LineSpec)` specify line type and color using any valid `LineSpec`.

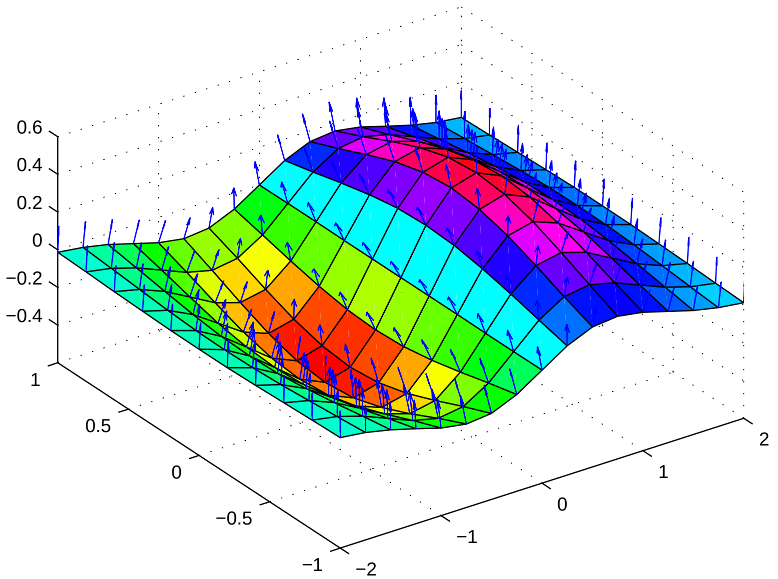
`quiver3(...,LineSpec,'filled')` fills markers specified by `LineSpec`.

`h = quiver3(...)` returns a vector of line handles.

Examples Plot the surface normals of the function $z = xe^{(-x^2 - y^2)}$.

```
[X,Y] = meshgrid(-2:0.25:2,-1:0.2:1);
Z = X.* exp(-X.^2 - Y.^2);
[U,V,W] = surfnorm(X,Y,Z);
quiver3(X,Y,Z,U,V,W,0.5);
hold on
surf(X,Y,Z);
colormap hsv
```

```
view(-35,45)
axis ([-2 2 -1 1 -.6 .6])
hold off
```



See Also

`axis`, `contour`, `LineSpec`, `plot`, `plot3`, `quiver`, `surfnorm`, `view`

Purpose QZ factorization for generalized eigenvalues

Syntax

```
[AA,BB,Q,Z,] = qz(A,B)
[AA,BB,Q,Z,V,W] = qz(A,B)
qz(A,B,flag)
```

Description The `qz` function gives access to intermediate results in the computation of generalized eigenvalues.

`[AA,BB,Q,Z] = qz(A,B)` for square matrices `A` and `B`, produces upper quasitriangular matrices `AA` and `BB`, and unitary matrices `Q` and `Z` such that `Q*A*Z = AA`, and `Q*B*Z = BB`. For complex matrices, `AA` and `BB` are triangular.

`[AA,BB,Q,Z,V,W] = qz(A,B)` also produces matrices `V` and `W` whose columns are generalized eigenvectors.

`qz(A,B,flag)` for real matrices `A` and `B`, produces one of two decompositions depending on the value of `flag`:

`'complex'`	Produces a possibly complex decomposition with a triangular `AA`. For compatibility with earlier versions, `'complex'` is the default.
`'real'`	Produces a real decomposition with a quasitriangular `AA`, containing 1-by-1 and 2-by-2 blocks on its diagonal.

If `AA` is triangular, the diagonal elements of `AA` and `BB`,

```
alpha = diag(AA)
beta = diag(BB)
```

are the generalized eigenvalues that satisfy

```
A*V*diag(beta) = B*V*diag(alpha)
diag(beta)*W'*A = diag(alpha)*W'*B
```

The eigenvalues produced by

```
lambda = eig(A,B)
```

are the element-wise ratios of alpha and beta.

```
lambda = alpha ./ beta
```

If AA is not triangular, it is necessary to further reduce the 2-by-2 blocks to obtain the eigenvalues of the full system.

Algorithm

For real QZ on real A and real B, eig uses the LAPACK DGGES routine. If you request the fifth output V, eig also uses DTGEVC.

For complex QZ on real or complex A and B, eig uses the LAPACK ZGGES routine. If you request the fifth output V, eig also uses ZTGEVC.

See Also

eig

References

[1] Anderson, E., Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, *LAPACK User's Guide* (http://www.netlib.org/lapack/lug/lapack_lug.html), Third Edition, SIAM, Philadelphia, 1999.

Purpose Uniformly distributed random numbers and arrays

Syntax

```
Y = rand(n)
Y = rand(m,n)
Y = rand([m n])
Y = rand(m,n,p,...)
Y = rand([m n p...])
Y = rand(size(A))
rand
s = rand('state')
```

Description The `rand` function generates arrays of random numbers whose elements are uniformly distributed in the interval (`0`,`1`).

`Y = rand(n)` returns an `n`-by-`n` matrix of random entries. An error message appears if `n` is not a scalar.

`Y = rand(m,n)` or `Y = rand([m n])` returns an `m`-by-`n` matrix of random entries.

`Y = rand(m,n,p,...)` or `Y = rand([m n p...])` generates random arrays.

`Y = rand(size(A))` returns an array of random entries that is the same size as `A`.

`rand`, by itself, returns a scalar whose value changes each time it's referenced.

`s = rand('state')` returns a 35-element vector containing the current state of the uniform generator. To change the state of the generator:

`rand('state',s)`	Resets the state to `s`.
`rand('state',0)`	Resets the generator to its initial state.
`rand('state',j)`	For integer `j`, resets the generator to its `j`-th state.
`rand('state',sum(100*clock))`	Resets it to a different state each time.

Examples

Example 1. `R = rand(3,4)` may produce

```
R =
   0.2190    0.6793    0.5194    0.0535
   0.0470    0.9347    0.8310    0.5297
   0.6789    0.3835    0.0346    0.6711
```

This code makes a random choice between two equally probable alternatives.

```
if rand < .5
   'heads'
else
   'tails'
end
```

Example 2. Generate a uniform distribution of random numbers on a specified interval `[a,b]`. To do this, multiply the output of `rand` by `(b-a)` then add `a`. For example, to generate a 5-by-5 array of uniformly distributed random numbers on the interval `[10,50]`

```
a = 10; b = 50;
x = a + (b-a) * rand(5)
x =

   18.1106   10.6110   26.7460   43.5247   30.1125
   17.9489   39.8714   43.8489   10.7856   38.3789
   34.1517   27.8039   31.0061   37.2511   27.1557
   20.8875   47.2726   18.1059   25.1792   22.1847
   17.9526   28.6398   36.8855   43.2718   17.5861
```

See Also

`randn`, `randperm`, `sprand`, `sprandn`

Purpose Normally distributed random numbers and arrays

Syntax

```
Y = randn(n)
Y = randn(m,n)
Y = randn([m n])
Y = randn(m,n,p,...)
Y = randn([m n p...])
Y = randn(size(A))
randn
s = randn('state')
```

Description The `randn` function generates arrays of random numbers whose elements are normally distributed with mean 0, variance $\sigma^2 = 1$, and standard deviation $\sigma = 1$.

`Y = randn(n)` returns an `n`-by-`n` matrix of random entries. An error message appears if `n` is not a scalar.

`Y = randn(m,n)` or `Y = randn([m n])` returns an `m`-by-`n` matrix of random entries.

`Y = randn(m,n,p,...)` or `Y = randn([m n p...])` generates random arrays.

`Y = randn(size(A))` returns an array of random entries that is the same size as `A`.

`randn`, by itself, returns a scalar whose value changes each time it's referenced.

`s = randn('state')` returns a 2-element vector containing the current state of the normal generator. To change the state of the generator:

`randn('state',s)`	Resets the state to `s`.
`randn('state',0)`	Resets the generator to its initial state.
`randn('state',j)`	For integer `j`, resets the generator to its `j`th state.
`randn('state',sum(100*clock))`	Resets it to a different state each time.

Examples

Example 1. `R = randn(3,4)` may produce

```
R =
      1.1650    0.3516    0.0591    0.8717
      0.6268   -0.6965    1.7971   -1.4462
      0.0751    1.6961    0.2641   -0.7012
```

For a histogram of the `randn` distribution, see `hist`.

Example 2. Generate a random distribution with a specific mean and variance σ^2. To do this, multiply the output of `randn` by the standard deviation σ, and then add the desired mean. For example, to generate a 5-by-5 array of random numbers with a mean of .6 that are distributed with a variance of 0.1

```
x = .6 + sqrt(0.1) * randn(5)
x =

    0.8713    0.4735    0.8114    0.0927    0.7672
    0.9966    0.8182    0.9766    0.6814    0.6694
    0.0960    0.8579    0.2197    0.2659    0.3085
    0.1443    0.8251    0.5937    1.0475   -0.0864
    0.7806    1.0080    0.5504    0.3454    0.5813
```

See Also

`rand`, `randperm`, `sprand`, `sprandn`

Purpose Random permutation

Syntax

```
p = randperm(n)
```

Description `p = randperm(n)` returns a random permutation of the integers `1:n`.

Remarks The `randperm` function calls `rand` and therefore changes `rand`'s state.

Examples `randperm(6)` might be the vector

```
[3  2  6  4  1  5]
```

or it might be some other permutation of `1:6`.

See Also `permute`

rank

Purpose

Rank of a matrix

Syntax

```
k = rank(A)
k = rank(A,tol)
```

Description

The `rank` function provides an estimate of the number of linearly independent rows or columns of a full matrix.

`k = rank(A)` returns the number of singular values of `A` that are larger than the default tolerance, `max(size(A))*norm(A)*eps`.

`k = rank(A,tol)` returns the number of singular values of `A` that are larger than `tol`.

Remark

Use `sprank` to determine the structural rank of a sparse matrix.

Algorithm

There are a number of ways to compute the rank of a matrix. MATLAB uses the method based on the singular value decomposition, or SVD. The SVD algorithm is the most time consuming, but also the most reliable.

The `rank` algorithm is

```
s = svd(A);
tol = max(size(A))*s(1)*eps;
r = sum(s > tol);
```

See Also

`sprank`

References

[1] Anderson, E., Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, *LAPACK User's Guide* (`http://www.netlib.org/lapack/lug/lapack_lug.html`), Third Edition, SIAM, Philadelphia, 1999.

Purpose

Rational fraction approximation

Syntax

```
[N,D] = rat(X)
[N,D] = rat(X,tol)
rat(...)
S = rats(X,strlen)
S = rats(X)
```

Description

Even though all floating-point numbers are rational numbers, it is sometimes desirable to approximate them by simple rational numbers, which are fractions whose numerator and denominator are small integers. The `rat` function attempts to do this. Rational approximations are generated by truncating continued fraction expansions. The `rats` function calls `rat`, and returns strings.

`[N,D] = rat(X)` returns arrays `N` and `D` so that `N./D` approximates `X` to within the default tolerance, `1.e-6*norm(X(:),1)`.

`[N,D] = rat(X,tol)` returns `N./D` approximating `X` to within `tol`.

`rat(X)`, with no output arguments, simply displays the continued fraction.

`S = rats(X,strlen)` returns a string containing simple rational approximations to the elements of `X`. Asterisks are used for elements that cannot be printed in the allotted space, but are not negligible compared to the other elements in `X`. `strlen` is the length of each string element returned by the `rats` function. The default is `strlen = 13`, which allows 6 elements in 78 spaces.

`S = rats(X)` returns the same results as those printed by MATLAB with `format rat`.

Examples

Ordinarily, the statement

```
s = 1 - 1/2 + 1/3 - 1/4 + 1/5 - 1/6 + 1/7
```

produces

```
s =
   0.7595
```

However, with

```
format rat
```

or with

```
rats(s)
```

the printed result is

```
s =
   319/420
```

This is a simple rational number. Its denominator is 420, the least common multiple of the denominators of the terms involved in the original expression. Even though the quantity `s` is stored internally as a binary floating-point number, the desired rational form can be reconstructed.

To see how the rational approximation is generated, the statement `rat(s)`

produces

```
1 + 1/(-4 + 1/(-6 + 1/(-3 + 1/(-5))))
```

And the statement

```
[n,d] = rat(s)
```

produces

```
n = 319, d = 420
```

The mathematical quantity π is certainly not a rational number, but the MATLAB quantity `pi` that approximates it is a rational number. `pi` is the ratio of a large integer and 2^{52}:

```
14148475504056880/4503599627370496
```

However, this is not a simple rational number. The value printed for `pi` with `format rat`, or with `rats(pi)`, is

```
355/113
```

This approximation was known in Euclid's time. Its decimal representation is

```
3.14159292035398
```

and so it agrees with `pi` to seven significant figures. The statement

```
rat(pi)
```

produces

```
3 + 1/(7 + 1/(16))
```

This shows how the `355/113` was obtained. The less accurate, but more familiar approximation `22/7` is obtained from the first two terms of this continued fraction.

Algorithm

The `rat(X)` function approximates each element of `X` by a continued fraction of the form

$$\frac{n}{d} = d_1 + \cfrac{1}{d_2 + \cfrac{1}{\left(d_3 + \ldots + \cfrac{1}{d_k}\right)}}$$

The d s are obtained by repeatedly picking off the integer part and then taking the reciprocal of the fractional part. The accuracy of the approximation increases exponentially with the number of terms and is worst when `X = sqrt(2)`. For `x = sqrt(2)`, the error with `k` terms is about `2.68*(.173)^k`, so each additional term increases the accuracy by less than one decimal digit. It takes 21 terms to get full floating-point accuracy.

See Also

`format`

rbbox

Purpose Create rubberband box for area selection

Synopsis

```
rbbox
rbbox(initialRect)
rbbox(initialRect,fixedPoint)
rbbox(initialRect,fixedPoint,stepSize)
finalRect = rbbox(...)
```

Description

`rbbox` initializes and tracks a rubberband box in the current figure. It sets the initial rectangular size of the box to 0, anchors the box at the figure's `CurrentPoint`, and begins tracking from this point.

`rbbox(initialRect)` specifies the initial location and size of the rubberband box as `[x y width height]`, where `x` and `y` define the lower-left corner, and `width` and `height` define the size. `initialRect` is in the units specified by the current figure's `Units` property, and measured from the lower-left corner of the figure window. The corner of the box closest to the pointer position follows the pointer until `rbbox` receives a button-up event.

`rbbox(initialRect,fixedPoint)` specifies the corner of the box that remains fixed. All arguments are in the units specified by the current figure's `Units` property, and measured from the lower-left corner of the figure window. `fixedPoint` is a two-element vector, `[x y]`. The tracking point is the corner diametrically opposite the anchored corner defined by `fixedPoint`.

`rbbox(initialRect,fixedPoint,stepSize)` specifies how frequently the rubberband box is updated. When the tracking point exceeds `stepSize` figure units, `rbbox` redraws the rubberband box. The default stepsize is `1`.

`finalRect = rbbox(...)` returns a four-element vector, `[x y width height]`, where `x` and `y` are the *x* and *y* components of the lower-left corner of the box, and `width` and `height` are the dimensions of the box.

Remarks

`rbbox` is useful for defining and resizing a rectangular region:

- For box definition, `initialRect` is `[x y 0 0]`, where `(x,y)` is the figure's `CurrentPoint`.

- For box resizing, `initialRect` defines the rectangular region that you resize (e.g., a legend). `fixedPoint` is the corner diametrically opposite the tracking point.

`rbbox` returns immediately if a button is not currently pressed. Therefore, you use `rbbox` with `waitforbuttonpress` so that the mouse button is down when `rbbox` is called. `rbbox` returns when you release the mouse button.

Examples

Assuming the current view is `view(2)`, use the current axes' `CurrentPoint` property to determine the extent of the rectangle in dataspace units:

```
k = waitforbuttonpress;

point1 = get(gca,'CurrentPoint');    % button down detected
finalRect = rbbox;                   % return figure units
point2 = get(gca,'CurrentPoint');    % button up detected

point1 = point1(1,1:2);              % extract x and y
point2 = point2(1,1:2);

p1 = min(point1,point2);             % calculate locations
offset = abs(point1-point2);         % and dimensions

x = [p1(1) p1(1)+offset(1) p1(1)+offset(1) p1(1) p1(1)];
y = [p1(2) p1(2) p1(2)+offset(2) p1(2)+offset(2) p1(2)];

hold on
axis manual
plot(x,y)                            % redraw in dataspace units
```

See Also

`axis`, `dragrect`, `waitforbuttonpress`

rcond

Purpose Matrix reciprocal condition number estimate

Syntax

```
c = rcond(A)
```

Description **`c = rcond(A)`** returns an estimate for the reciprocal of the condition of `A` in 1-norm using the LAPACK condition estimator. If `A` is well conditioned, `rcond(A)` is near 1.0. If `A` is badly conditioned, `rcond(A)` is near 0.0. Compared to `cond`, `rcond` is a more efficient, but less reliable, method of estimating the condition of a matrix.

Algorithm `rcond` uses LAPACK routines to compute the estimate of the reciprocal condition number:

Matrix	Routine
Real	`DLANGE`, `DGETRF`, `DGECON`
Complex	`ZLANGE`, `ZGETRF`, `ZGECON`

See Also `cond`, `condest`, `norm`, `normest`, `rank`, `svd`

References

[1] Anderson, E., Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, *LAPACK User's Guide* (`http://www.netlib.org/lapack/lug/lapack_lug.html`), Third Edition, SIAM, Philadelphia, 1999.

Purpose Read data asynchronously from the device

Syntax

```
readasync(obj)
readasync(obj,size)
```

Arguments

`obj`	A serial port object.
`size`	The number of bytes to read from the device.

Description

`readasync(obj)` initiates an asynchronous read operation.

`readasync(obj,size)` asynchronously reads, at most, the number of bytes given by `size`. If `size` is greater than the difference between the `InputBufferSize` property value and the `BytesAvailable` property value, an error is returned.

Remarks

Before you can read data, you must connect `obj` to the device with the `fopen` function. A connected serial port object has a `Status` property value of `open`. An error is returned if you attempt to perform a read operation while `obj` is not connected to the device.

You should use `readasync` only when you configure the `ReadAsyncMode` property to `manual`. `readasync` is ignored if used when `ReadAsyncMode` is `continuous`.

The `TransferStatus` property indicates if an asynchronous read or write operation is in progress. You can write data while an asynchronous read is in progress since serial ports have separate read and write pins. You can stop asynchronous read and write operations with the `stopasync` function.

You can monitor the amount of data stored in the input buffer with the `BytesAvailable` property. Additionally, you can use the `BytesAvailableFcn` property to execute an M-file callback function when the terminator or the specified amount of data is read.

Rules for Completing an Asynchronous Read Operation

An asynchronous read operation with `readasync` completes when one of these conditions is met:

- The terminator specified by the `Terminator` property is read.

- The time specified by the `Timeout` property passes.
- The specified number of bytes is read.
- The input buffer is filled (if `size` is not specified).

Since `readasync` checks for the terminator, this function can be slow. To increase speed, you may want to configure `ReadAsyncMode` to `continuous` and continuously return data to the input buffer as soon as it is available from the device.

Example

This example creates the serial port object `s`, connects `s` to a Tektronix TDS 210 oscilloscope, configures `s` to read data asynchronously only if `readasync` is issued, and configures the instrument to return the peak-to-peak value of the signal on channel 1.

```
s = serial('COM1');
fopen(s)
s.ReadAsyncMode = 'manual';
fprintf(s,'Measurement:Meas1:Source CH1')
fprintf(s,'Measurement:Meas1:Type Pk2Pk')
fprintf(s,'Measurement:Meas1:Value?')
```

Begin reading data asynchronously from the instrument using `readasync`. When the read operation is complete, return the data to the MATLAB workspace using `fscanf`.

```
readasync(s)
s.BytesAvailable
ans =
     15
out = fscanf(s)
out =
2.0399999619E0
fclose(s)
```

See Also

Functions

`fopen`, `stopasync`

Properties

`BytesAvailable`, `BytesAvailableFcn`, `ReadAsyncMode`, `Status`, `TransferStatus`

Purpose Real part of complex number

Syntax `X = real(Z)`

Description `X = real(Z)` returns the real part of the elements of the complex array `Z`.

Examples `real(2+3*i)` is `2`.

See Also `abs`, `angle`, `conj`, `i`, `j`, `imag`

realmax

Purpose Largest positive floating-point number

Syntax

```
n = realmax
```

Description `n = realmax` returns the largest floating-point number representable on a particular computer. Anything larger overflows.

Examples `realmax` is one bit less than 2^{1024} or about `1.7977e+308`.

Algorithm The `realmax` function is equivalent to `pow2(2-eps,maxexp)`, where `maxexp` is the largest possible floating-point exponent.

Execute `type realmax` to see `maxexp` for various computers.

See Also `eps`, `realmin`

Purpose Smallest positive floating-point number

Syntax

```
n = realmin
```

Description `n = realmin` returns the smallest positive normalized floating-point number on a particular computer. Anything smaller underflows or is an IEEE "denormal."

Examples On machines with IEEE floating-point format, `realmin` is `2^(-1022)` or about `2.2251e-308`.

Algorithm The `realmin` function is equivalent to `pow2(1,minexp)` where `minexp` is the smallest possible floating-point exponent.

Execute `type realmin` to see `minexp` for various computers.

See Also `eps`, `realmax`

record

Purpose Record data and event information to a file

Syntax

```
record(obj)
record(obj,'switch')
```

Arguments

`obj`	A serial port object.
`'switch'`	Switch recording capabilities on or off.

Description

`record(obj)` toggles the recording state for `obj`.

`record(obj,'switch')` initiates or terminates recording for `obj`. *`switch`* can be `on` or `off`. If *`switch`* is `on`, recording is initiated. If *`switch`* is `off`, recording is terminated.

Remarks

Before you can record information to disk, `obj` must be connected to the device with the `fopen` function. A connected serial port object has a `Status` property value of `open`. An error is returned if you attempt to record information while `obj` is not connected to the device. Each serial port object must record information to a separate file. Recording is automatically terminated when `obj` is disconnected from the device with `fclose`.

The `RecordName` and `RecordMode` properties are read-only while `obj` is recording, and must be configured before using `record`.

For a detailed description of the record file format and the properties associated with recording data and event information to a file, refer to “Debugging: Recording Information to Disk.”

Example

This example creates the serial port object `s`, connects `s` to the device, configures `s` to record information to a file, writes and reads text data, and then disconnects `s` from the device.

```
s = serial('COM1');
fopen(s)
s.RecordDetail = 'verbose';
s.RecordName = 'MySerialFile.txt';
record(s,'on')
fprintf(s,'*IDN?')
out = fscanf(s);
```

```
record(s,'off')
fclose(s)
```

See Also

Functions

fclose, fopen

Properties

RecordDetail, RecordMode, RecordName, RecordStatus, Status

rectangle

Purpose

Create a 2-D rectangle object

Syntax

```
rectangle
rectangle('Position',[x,y,w,h])
rectangle(...,'Curvature',[x,y])
h = rectangle(...)
```

Description

`rectangle` draws a rectangle with `Position` `[0,0,1,1]` and `Curvature` `[0,0]` (i.e., no curvature).

`rectangle('Position',[x,y,w,h])` draws the rectangle from the point `x,y` and having a width of `w` and a height of `h`. Specify values in axes data units.

Note that, to display a rectangle in the specified proportions, you need to set the axes data aspect ratio so that one unit is of equal length along both the x and y axes. You can do this with the command `axis equal` or `daspect([1,1,1])`.

`rectangle(...,'Curvature',[x,y])` specifies the curvature of the rectangle sides, enabling it to vary from a rectangle to an ellipse. The horizontal curvature `x` is the fraction of width of the rectangle that is curved along the top and bottom edges. The vertical curvature `y` is the fraction of the height of the rectangle that is curved along the left and right edges.

The values of `x` and `y` can range from `0` (no curvature) to `1` (maximum curvature). A value of `[0,0]` creates a rectangle with square sides. A value of `[1,1]` creates an ellipse. If you specify only one value for `Curvature`, then the same length (in axes data units) is curved along both horizontal and vertical sides. The amount of curvature is determined by the shorter dimension.

`h = rectangle(...)` returns the handle of the rectangle object created.

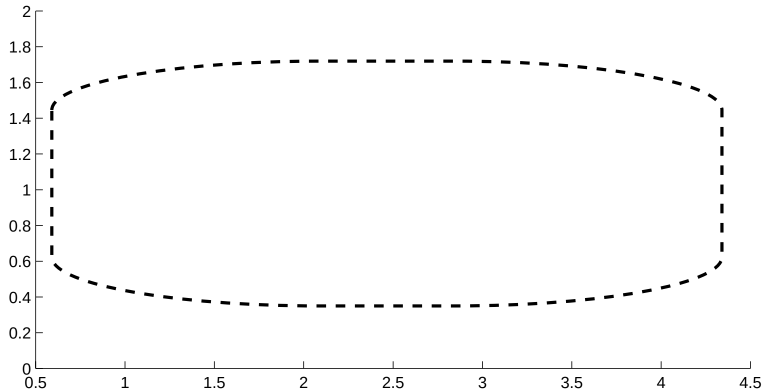
Remarks

Rectangle objects are 2-D and can be drawn in an axes only if the view is `[0 90]` (i.e., `view(2)`). Rectangles are children of axes and are defined in coordinates of the axes data.

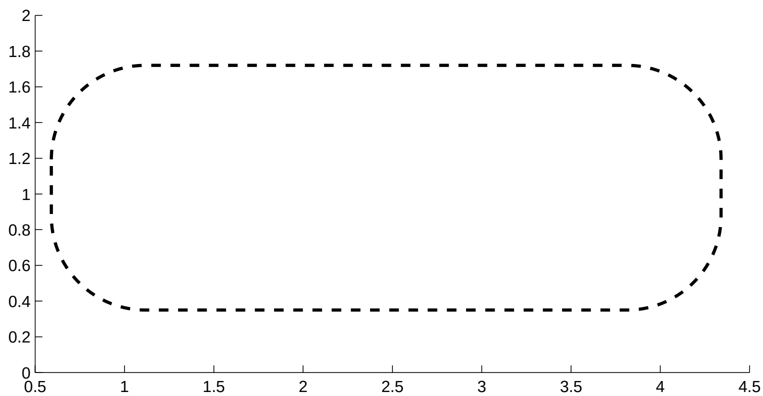
Examples

This example sets the data aspect ratio to `[1,1,1]` so that the rectangle displays in the specified proportions (`daspect`). Note that the horizontal and vertical curvature can be different. Also, note the effects of using a single value for `Curvature`.

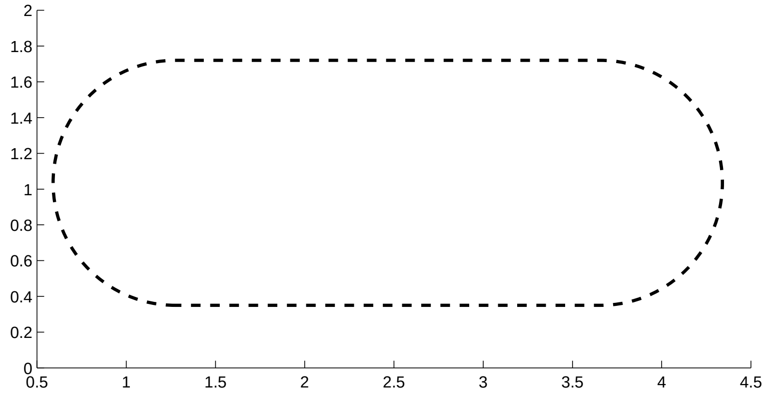
```
rectangle('Position',[0.59,0.35,3.75,1.37],...
          'Curvature',[0.8,0.4],...
         'LineWidth',2,'LineStyle','--')
daspect([1,1,1])
```



Specifying a single value of `[0.4]` for `Curvature` produces:

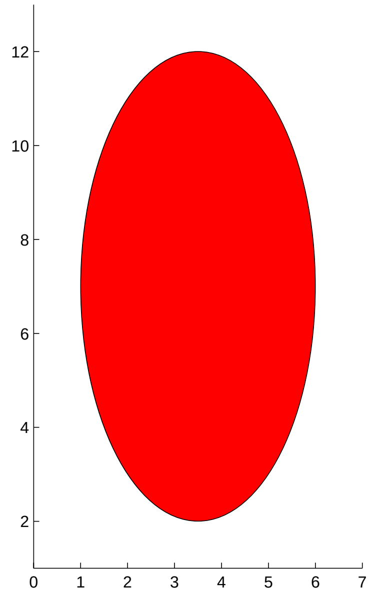


A `Curvature` of `[1]` produces a rectangle with the shortest side completely round:



This example creates an ellipse and colors the face red.

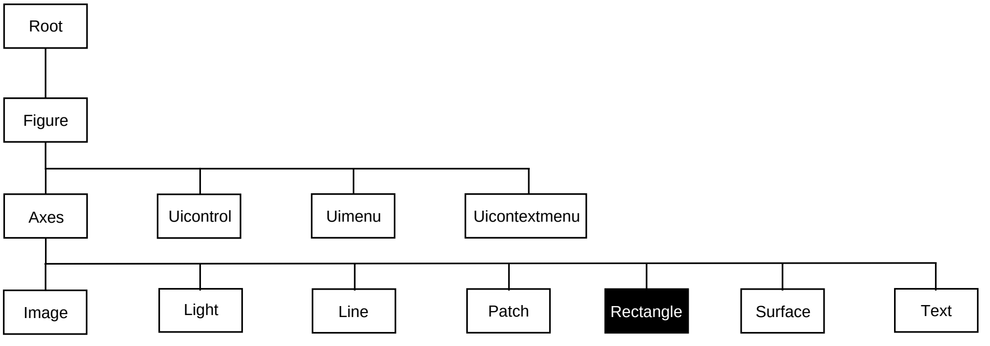
```
rectangle('Position',[1,2,5,10],'Curvature',[1,1],...
          'FaceColor’,'r')
daspect([1,1,1])
xlim([0,7])
ylim([1,13])
```



See Also

`line`, `patch`, `plot`, `plot3`, `set`, `text`, rectangle properties

Object Hierarchy



Setting Default Properties

You can set default rectangle properties on the axes, figure, and root levels.

```
set(0,'DefaultRectangleProperty',PropertyValue...)
set(gcf,'DefaultRectangleProperty',PropertyValue...)
set(gca,'DefaultRectangleProperty',PropertyValue...)
```

Where *Property* is the name of the rectangle property whose default value you want to set and `PropertyValue` is the value you are specifying. Use `set` and `get` to access the surface properties.

Property List

The following table lists all rectangle properties and provides a brief description of each. The property name links take you to an expanded description of the properties.

Property Name	Property Description	Property Value
Defining the Rectangle Object		
`Curvature`	Degree of horizontal and vertical curvature	Value: two-element vector with values between `0` and `1` Default: `[0,0]`
`EraseMode`	Method of drawing and erasing the rectangle (useful for animation)	Values: `normal`, `none`, `xor`, `background` Default: `normal`
`EdgeColor`	Color of rectangle edges	Value: `ColorSpec` or `none` Default: `ColorSpec [0,0,0]`
`FaceColor`	Color of rectangle interior	Value: `ColorSpec` or `none` Default: `none`
`LineStyle`	Line style of edges	Values: –, ––, :, –., `none` Default: –
`LineWidth`	Width of edge lines in points	Value: scalar Default: `0.5` points
`Position`	Location and width and height of rectangle	Value: [x,y,width,height] Default: `[0,0,1,1]`

Property Name	Property Description	Property Value
General Information About Rectangle Objects		
`Children`	Rectangle objects have no children	
`Parent`	Axes object	Value: handle of axes
`Selected`	Indicate if the rectangle is in a "selected" state.	Value: `on`, `off` Default: `off`
`Tag`	User-specified label	Value: any string Default: `''` (empty string)
`Type`	The type of graphics object (read only)	Value: the string `'rectangle'`
`UserData`	User-specified data	Value: any matrix Default: `[]` (empty matrix)
Properties Related to Callback Routine Execution		
`BusyAction`	Specify how to handle callback routine interruption	Value: `cancel`, `queue` Default: `queue`
`ButtonDownFcn`	Define a callback routine that executes when a mouse button is pressed on over the rectangle	Value: string Default: `''` (empty string)
`CreateFcn`	Define a callback routine that executes when a rectangle is created	Value: string Default: `''` (empty string)
`DeleteFcn`	Define a callback routine that executes when the rectangle is deleted (via `close` or `delete`)	Values: string Default: `''` (empty string)
`Interruptible`	Determine if callback routine can be interrupted	Values: `on`, `off` Default: `on` (can be interrupted)
`UIContextMenu`	Associate a context menu with the rectangle	Values: handle of a Uicontextmenu

Property Name	Property Description	Property Value
Controlling Access to Objects		
`HandleVisibility`	Determines if and when the rectangle's handle is visible to other functions	Values: `on`, `callback`, `off` Default: `on`
`HitTest`	Determines if the rectangle can become the current object (see the Figure `CurrentObject` property)	Values: `on`, `off` Default: `on`
Controlling the Appearance		
`Clipping`	Clipping to axes rectangle	Values: `on`, `off` Default: `on`
`SelectionHighlight`	Highlight rectangle when selected (`Selected` property set to `on`)	Values: `on`, `off` Default: `on`
`Visible`	Make the rectangle visible or invisible	Values: `on`, `off` Default: `on`

Modifying Properties

You can set and query graphics object properties in two ways:

- The Property Editor is an interactive tool that enables you to see and change object property values.
- The `set` and `get` commands enable you to set and query the values of properties

To change the default value of properties see Setting Default Property Values.

Rectangle Property Descriptions

This section lists property names along with the type of values each accepts. Curly braces { } enclose default values.

`BusyAction` `cancel | {queue}`

Callback routine interruption. The `BusyAction` property enables you to control how MATLAB handles events that potentially interrupt executing callback routines. If there is a callback routine executing, subsequently invoked callback routes always attempt to interrupt it. If the `Interruptible` property of the object whose callback is executing is set to `on` (the default), then interruption occurs at the next point where the event queue is processed. If the `Interruptible` property is `off`, the `BusyAction` property (of the object owning the executing callback) determines how MATLAB handles the event. The choices are:

- `cancel` – discard the event that attempted to execute a second callback routine.
- `queue` – queue the event that attempted to execute a second callback routine until the current callback finishes.

`ButtonDownFcn` string

Button press callback routine. A callback routine that executes whenever you press a mouse button while the pointer is over the rectangle object. Define this routine as a string that is a valid MATLAB expression or the name of an M-file. The expression executes in the MATLAB workspace.

`Children` vector of handles

The empty matrix; rectangle objects have no children.

Clipping {on} | off

Clipping mode. MATLAB clips rectangles to the axes plot box by default. If you set `Clipping` to `off`, rectangles display outside the axes plot box. This can occur if you create a rectangle, set `hold` to `on`, freeze axis scaling (`axis manual`), and then create a larger rectangle.

CreateFcn string

Callback routine executed during object creation. This property defines a callback routine that executes when MATLAB creates a rectangle object. You must define this property as a default value for rectangles. For example, the statement,

```
set(0,'DefaultRectangleCreateFcn',...
    'set(gca,''DataAspectRatio'',[1,1,1])')
```

defines a default value on the root level that sets the axes `DataAspectRatio` whenever you create a rectangle object. MATLAB executes this routine after setting all rectangle properties. Setting this property on an existing rectangle object has no effect.

The handle of the object whose `CreateFcn` is being executed is accessible only through the root `CallbackObject` property, which you can query using `gcbo`.

Curvature one- or two-element vector `[x,y]`

Amount of horizontal and vertical curvature. This property specifies the curvature of the property sides, which enables the shape of the rectangle to vary from rectangular to ellipsoidal. The horizontal curvature `x` is the fraction of width of the rectangle that is curved along the top and bottom edges. The vertical curvature `y` is the fraction of the height of the rectangle that is curved along the left and right edges.

The values of `x` and `y` can range from `0` (no curvature) to `1` (maximum curvature). A value of `[0,0]` creates a rectangle with square sides. A value of `[1,1]` creates an ellipse. If you specify only one value for `Curvature`, then the same length (in axes data units) is curved along both horizontal and vertical sides. The amount of curvature is determined by the shorter dimension.

DeleteFcn string

Delete rectangle callback routine. A callback routine that executes when you delete the rectangle object (e.g., when you issue a `delete` command or clear the axes or figure). MATLAB executes the routine before deleting the object's properties so these values are available to the callback routine.

The handle of the object whose `DeleteFcn` is being executed is accessible only through the root `CallbackObject` property, which you can query using `gcbo`.

`EdgeColor` `{ColorSpec} | none`

Color of the rectangle edges. This property specifies the color of the rectangle edges as a color or specifies that no edges be drawn.

`EraseMode` `{normal} | none | xor | background`

Erase mode. This property controls the technique MATLAB uses to draw and erase rectangle objects. Alternative erase modes are useful for creating animated sequences, where control of the way individual objects redraw is necessary to improve performance and obtain the desired effect.

- `normal` (the default) – Redraw the affected region of the display, performing the three-dimensional analysis necessary to ensure that all objects are rendered correctly. This mode produces the most accurate picture, but is the slowest. The other modes are faster, but do not perform a complete redraw and are therefore less accurate.
- `none` – Do not erase the rectangle when it is moved or destroyed. While the object is still visible on the screen after erasing with `EraseMode none`, you cannot print it because MATLAB stores no information about its former location.
- `xor` – Draw and erase the rectangle by performing an exclusive OR (XOR) with the color of the screen beneath it. This mode does not damage the color of the objects beneath the rectangle. However, the rectangle's color depends on the color of whatever is beneath it on the display.
- `background` – Erase the rectangle by drawing it in the Axes' background `Color`, or the Figure background `Color` if the Axes `Color` is set to `none`. This damages objects that are behind the erased rectangle, but rectangles are always properly colored.

Printing with Non-normal Erase Modes.

MATLAB always prints Figures as if the `EraseMode` of all objects is `normal`. This means graphics objects created with `EraseMode` set to `none`, `xor`, or `background` can look different on screen than on paper. On screen, MATLAB may mathematically combine layers of colors (e.g., XORing a pixel color with that of the pixel behind it) and ignore three-dimensional sorting to obtain greater rendering speed. However, these techniques are not applied to the printed output.

You can use the MATLAB `getframe` command or other screen capture application to create an image of a Figure containing non-normal mode objects.

`FaceColor` `ColorSpec | {none}`

Color of rectangle face. This property specifies the color of the rectangle face, which is not colored by default.

`HandleVisibility` `{on} | callback | off`

Control access to object's handle by command-line users and GUIs. This property determines when an object's handle is visible in its parent's list of children. `HandleVisibility` is useful for preventing command-line users from accidentally drawing into or deleting a figure that contains only user interface devices (such as a dialog box).

Handles are always visible when `HandleVisibility` is `on`.

Setting `HandleVisibility` to `callback` causes handles to be visible from within callback routines or functions invoked by callback routines, but not from within functions invoked from the command line. This provides a means to protect GUIs from command-line users, while allowing callback routines to have complete access to object handles.

Setting `HandleVisibility` to `off` makes handles invisible at all times. This may be necessary when a callback routine invokes a function that might potentially damage the GUI (such as evaling a user-typed string), and so temporarily hides its own handles during the execution of that function.

When a handle is not visible in its parent's list of children, it cannot be returned by functions that obtain handles by searching the object hierarchy or querying handle properties. This includes `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close`.

When a handle's visibility is restricted using `callback` or `off`, the object's handle does not appear in its parent's `Children` property, figures do not appear in the root's `CurrentFigure` property, objects do not appear in the root's

`CallbackObject` property or in the figure's `CurrentObject` property, and Axes do not appear in their parent's `CurrentAxes` property.

You can set the Root `ShowHiddenHandles` property to `on` to make all handles visible, regardless of their `HandleVisibility` settings (this does not affect the values of the `HandleVisibility` properties).

Handles that are hidden are still valid. If you know an object's handle, you can `set` and `get` its properties, and pass it to any function that operates on handles.

`HitTest` `{on} | off`

Selectable by mouse click. `HitTest` determines if the rectangle can become the current object (as returned by the `gco` command and the figure `CurrentObject` property) as a result of a mouse click on the rectangle. If `HitTest` is `off`, clicking on the rectangle selects the object below it (which may be the axes containing it).

`Interruptible` `{on} | off`

Callback routine interruption mode. The `Interruptible` property controls whether a rectangle callback routine can be interrupted by subsequently invoked callback routines. Only callback routines defined for the `ButtonDownFcn` are affected by the `Interruptible` property. MATLAB checks for events that can interrupt a callback routine only when it encounters a `drawnow`, `figure`, `getframe`, or `pause` command in the routine.

`LineStyle` `{–} | –– | : | –. | none`

Line style. This property specifies the line style of the edges. The available line styles are:

Symbol	Line Style
–	solid line (default)
––	dashed line
:	dotted line
–.	dash-dot line
none	no line

LineWidth scalar

The width of the rectangle object. Specify this value in points (1 point = $^{1}/_{72}$ inch). The default `LineWidth` is 0.5 points.

Parent handle

rectangle's parent. The handle of the rectangle object's parent axes. You can move a rectangle object to another axes by changing this property to the new axes handle.

Position four-element vecotr `[x,y,width,height]`

Location and size of rectangle. This property specifies the location and size of the rectangle in the data units of the axes. The point defined by `x`, `y` specifies one corner of the rectangle, and `width` and `height` define the size in units along the x and y axes respecitvely.

Selected `on | off`

Is object selected? When this property is `on` MATLAB displays selection handles if the `SelectionHighlight` property is also `on`. You can, for example, define the `ButtonDownFcn` to set this property, allowing users to select the object with the mouse.

SelectionHighlight `{on} | off`

Objects highlight when selected. When the `Selected` property is on, MATLAB indicates the selected state by drawing handles at each vertex. When `SelectionHighlight` is `off`, MATLAB does not draw the handles.

Tag string

User-specified object label. The `Tag` property provides a means to identify graphics objects with a user-specified label. This is particularly useful when constructing interactive graphics programs that would otherwise need to define object handles as global variables or pass them as arguments between callback routines. You can define `Tag` as any string.

Type string (read only)

Class of graphics object. For rectangle objects, `Type` is always the string `'rectangle'`.

UIContextMenu handle of a uicontextmenu object

Associate a context menu with the rectangle. Assign this property the handle of a uicontextmenu object created in the same figure as the rectangle. Use the `uicontextmenu` function to create the context menu. MATLAB displays the context menu whenever you right-click over the rectangle.

UserData matrix

User-specified data. Any data you want to associate with the rectangle object. MATLAB does not use this data, but you can access it using the `set` and `get` commands.

Visible `{on} | off`

rectangle visibility. By default, all rectangles are visible. When set to `off`, the rectangle is not visible, but still exists and you can `get` and `set` its properties.

rectint

Purpose Rectangle intersection area.

Syntax

```
area = rectint(A,B)
```

Description

`area = rectint(A,B)` returns the area of intersection of the rectangles specified by position vectors `A` and `B`.

If `A` and `B` each specify one rectangle, the output `area` is a scalar.

`A` and `B` can also be matrices, where each row is a position vector. `area` is then a matrix giving the intersection of all rectangles specified by `A` with all the rectangles specified by `B`. That is, if `A` is `n`-by-4 and `B` is `m`-by-4, then `area` is an `n`-by-`m` matrix where `area(i,j)` is the intersection area of the rectangles specified by the `i`th row of `A` and the `j`th row of `B`.

Note A position vector is a four-element vector `[x,y,width,height]`, where the point defined by `x` and `y` specifies one corner of the rectangle, and `width` and `height` define the size in units along the x and y axes respectively.

See Also `polyarea`

Purpose Reduce the number of patch faces

Syntax

```
reducepatch(p,r)
nfv = reducepatch(p,r)
nfv = reducepatch(fv,r)
reducepatch(...,'fast')
reducepatch(...,'verbose')
nfv = reducepatch(f,v,r)
[nf,nv] = reducepatch(...)
```

Description

`reducepatch(p,r)` reduces the number of faces of the patch identified by handle `p`, while attempting to preserve the overall shape of the original object. MATLAB interprets the reduction factor `r` in one of two ways depending on its value:

- If `r` is less than 1, `r` is interpreted as a fraction of the original number of faces. For example, if you specify `r` as 0.2, then the number of faces is reduced to 20% of the number in the original patch.
- If `r` is greater than or equal to 1, then `r` is the target number of faces. For example, if you specify `r` as 400, then the number of faces is reduced until there are 400 faces remaining.

`nfv = reducepatch(p,r)` returns the reduced set of faces and vertices but does not set the `Faces` and `Vertices` properties of patch `p`. The struct `nfv` contains the faces and vertices after reduction.

`nfv = reducepatch(fv,r)` performs the reduction on the faces and vertices in the struct `fv`.

`nfv = reducepatch(p)` or `nfv = reducepatch(fv)` uses a reduction value of `0.5`.

`reducepatch(...,'fast')` assumes the vertices are unique and does not compute shared vertices.

`reducepatch(...,'verbose')` prints progress messages to the command window as the computation progresses.

`nfv = reducepatch(f,v,r)` performs the reduction on the faces in `f` and the vertices in `v`.

`[nf,nv] = reducepatch(...)` returns the faces and vertices in the arrays `nf` and `nv`.

Remarks

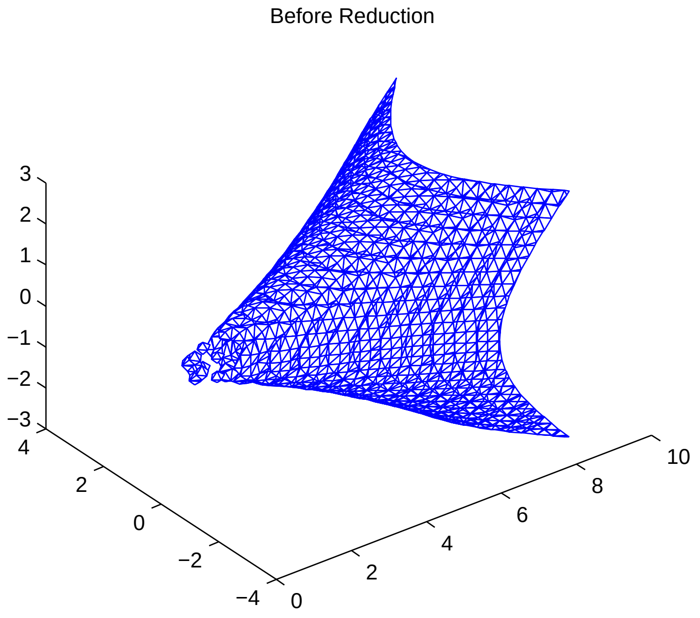
If the patch contains nonshared vertices, MATLAB computes shared vertices before reducing the number of faces. If the faces of the patch are not triangles, MATLAB triangulates the faces before reduction. The faces returned are always defined as triangles.

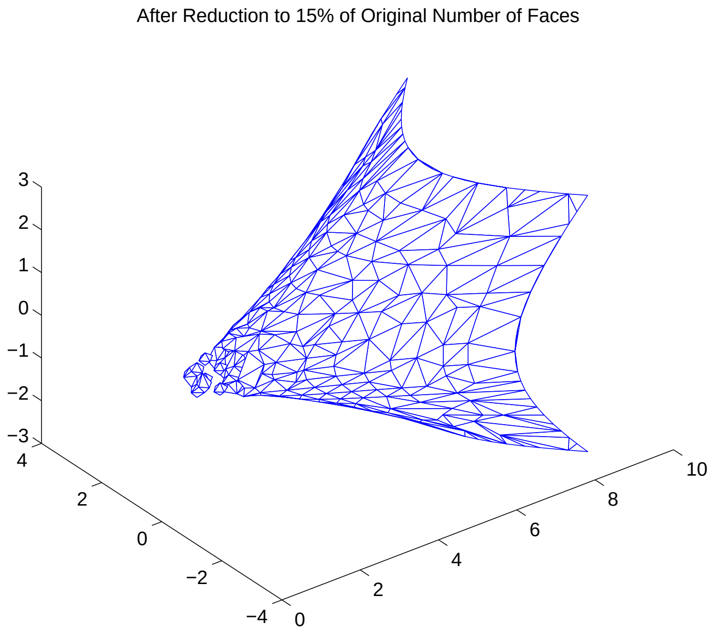
The number of output triangles may not be exactly the number specified with the reduction factor argument (`r`), particularly if the faces of the original patch are not triangles.

Examples

This example illustrates the effect of reducing the number of faces to only 15% of the original value.

```
[x,y,z,v] = flow;
p = patch(isosurface(x,y,z,v,-3));
set(p,'facecolor','w','EdgeColor','b');
daspect([1,1,1])
view(3)
figure;
h = axes;
p2 = copyobj(p,h);
reducepatch(p2,0.15)
daspect([1,1,1])
view(3)
```


Before Reduction
3
2
1
0
−1
−2
−3
4
2
0
−2
−4
0
2
4
6
8
10



See Also `isosurface`, `isocaps`, `isonormals`, `smooth3`, `subvolume`, `reducevolume`

Purpose

Reduce the number of elements in a volume data set

Syntax

```
[nx,ny,nz,nv] = reducevolume(X,Y,Z,V,[Rx,Ry,Rz])
[nx,ny,nz,nv] = reducevolume(V,[Rx,Ry,Rz])
nv = reducevolume(...)
```

Description

`[nx,ny,nz,nv] = reducevolume(X,Y,Z,V,[Rx,Ry,Rz])` reduces the number of elements in the volume by retaining every `Rx`th element in the x direction, every `Ry`th element in the y direction, and every `Rz`th element in the z direction. If a scalar `R` is used to indicate the amount or reduction instead of a 3-element vector, MATLAB assumes the reduction to be `[R R R]`.

The arrays `X`, `Y`, and `Z` define the coordinates for the volume `V`. The reduced volume is returned in `nv` and the coordinates of the reduced volume are returned in `nx`, `ny`, and `nz`.

`[nx,ny,nz,nv] = reducevolume(V,[Rx,Ry,Rz])` assumes the arrays `X`, `Y`, and `Z` are defined as `[X,Y,Z] = meshgrid(1:n,1:m,1:p)` where `[m,n,p] = size(V)`.

`nv = reducevolume(...)` returns only the reduced volume.

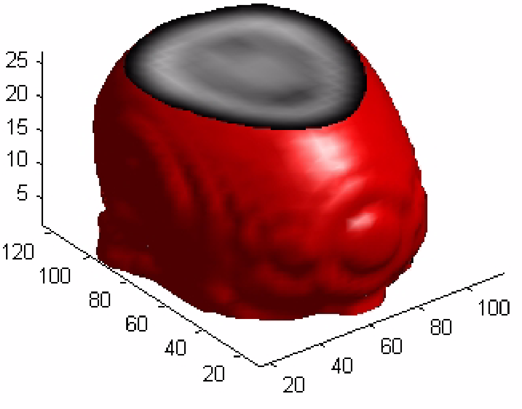
Examples

This example uses a data set that is a collection of MRI slices of a human skull. This data is processed in a variety of ways:

- The 4-D array is squeezed (`squeeze`) into three dimensions and then reduced (`reducevolume`) so that what remains is every 4th element in the x and y directions and every element in the z direction.
- The reduced data is smoothed (`smooth3`).
- The outline of the skull is an isosurface generated as a patch (`p1`) whose vertex normals are recalculated to improve the appearance when lighting is applied (`patch`, `isosurface`, `isonormals`).
- A second patch (`p2`) with an interpolated face color draws the end caps (`FaceColor`, `isocaps`).
- The view of the object is set (`view`, `axis`, `daspect`).
- A 100-element grayscale colormap provides coloring for the end caps (`colormap`).

- Adding a light to the right of the camera illuminates the object (`camlight`, `lighting`).

```
load mri
D = squeeze(D);
[x,y,z,D] = reducevolume(D,[4,4,1]);
D = smooth3(D);
p1 = patch(isosurface(x,y,z,D, 5,'verbose'),...
    'FaceColor','red','EdgeColor','none');
isonormals(x,y,z,D,p1);

p2 = patch(isocaps(x,y,z,D, 5),...
    'FaceColor','interp','EdgeColor','none');
view(3); axis tight; daspect([1,1,.4])
colormap(gray(100))
camlight; lighting gouraud
```



See Also

`isosurface`, `isocaps`, `isonormals`, `smooth3`, `subvolume`, `reducepatch`

Purpose Redraw current figure

Syntax

```
refresh
refresh(h)
```

Description `refresh` erases and redraws the current figure.

`refresh(h)` redraws the figure identified by `h`.

rehash

Purpose

Refresh function and file system caches

Syntax

```
rehash
rehash path
rehash toolbox
rehash pathreset
rehash toolboxreset
rehash toolboxcache
```

Description

`rehash` performs the same refresh that is done whenever MATLAB completes a command and returns to its prompt. The `rehash` function rereads changed directories, refreshes the list of known classes, and, if there are any functions whose source files have changed since they were loaded into memory, `rehash` clears those loaded functions.

`rehash` **`path`** is the same as `rehash`, except that it unconditionally rereads all nontoolbox directories. This is the same as the behavior of `path(`**`path`**`)`.

`rehash` **`toolbox`** is the same as `rehash` **`path`**, except that it unconditionally rereads all directories, including all toolbox directories.

`rehash` **`pathreset`** is the same as `rehash` **`path`**, except that it also forces any shadowed functions to be replaced by any shadowing functions.

`rehash` **`toolboxreset`** is the same as `rehash` **`toolbox`**, except that it also forces any shadowed functions to be replaced by any shadowing functions.

`rehash` **`toolboxcache`** generates a new toolbox cache. To use this command, you must first enable toolbox caching on your system. You also need read and write access to the directory that holds the toolbox cache file.

See Also

`addpath`, `path`, `rmpath`

Purpose Releases an interface.

Syntax

```
release (a)
```

Arguments

`a`
Activex object that represents the interface to be released.

Description Release the interface and all resources used by the interface. Each interface handle must be released when you are finished manipulating its properties and invoking its methods. Once an interface has been released, it is no longer valid and subsequent ActiveX operations on the MATLAB object that represents that interface will result in errors.

Note Releasing the interface will not delete the control itself (see `delete`), since other interfaces on that object may still be active. See "Releasing Interfaces" in *MATLAB External Interfaces* for more information.

Example

```
release (a)
```

rem

Purpose

Remainder after division

Syntax

```
R = rem(X,Y)
```

Description

`R = rem(X,Y)` returns `X - fix(X./Y).*Y`, where `fix(X./Y)` is the integer part of the quotient, `X./Y`.

Remarks

So long as operands `X` and `Y` are of the same sign, the statement `rem(X,Y)` returns the same result as does `mod(X,Y)`. However, for positive `X` and `Y`,

```
rem(-x,y) = mod(-x,y)-y
```

The `rem` function returns a result that is between 0 and `sign(X)*abs(Y)`. If `Y` is zero, `rem` returns `NaN`.

Limitations

Arguments `X` and `Y` should be integers. Due to the inexact representation of floating-point numbers on a computer, real (or complex) inputs may lead to unexpected results.

See Also

`mod`

Purpose Replicate and tile an array

Syntax

```
B = repmat(A,m,n)
B = repmat(A,[m n])
B = repmat(A,[m n p...])
repmat(A,m,n)
```

Description

`B = repmat(A,m,n)` creates a large matrix `B` consisting of an `m`-by-`n` tiling of copies of `A`. The statement `repmat(A,n)` creates an `n`-by-`n` tiling.

`B = repmat(A,[m n])` accomplishes the same result as `repmat(A,m,n)`.

`B = repmat(A,[m n p...])` produces a multidimensional (`m`-by-`n`-by-`p`-by-...) array composed of copies of `A`. `A` may be multidimensional.

`repmat(A,m,n)` when `A` is a scalar, produces an `m`-by-`n` matrix filled with `A`'s value. This can be much faster than `a*ones(m,n)` when `m` or `n` is large.

Examples

In this example, `repmat` replicates 12 copies of the second-order identity matrix, resulting in a "checkerboard" pattern.

```
B = repmat(eye(2),3,4)

B =
     1     0     1     0     1     0     1     0
     0     1     0     1     0     1     0     1
     1     0     1     0     1     0     1     0
     0     1     0     1     0     1     0     1
     1     0     1     0     1     0     1     0
     0     1     0     1     0     1     0     1
```

The statement `N = repmat(NaN,[2 3])` creates a 2-by-3 matrix of `NaN`s.

reset

Purpose Reset graphics object properties to their defaults

Syntax

```
reset(h)
```

Description `reset(h)` resets all properties having factory defaults on the object identified by `h`. To see the list of factory defaults, use the statement,

```
get(0,'factory')
```

If `h` is a figure, MATLAB does not reset `Position`, `Units`, `PaperPosition`, and `PaperUnits`. If `h` is an axes, MATLAB does not reset `Position` and `Units`.

Examples `reset(gca)` resets the properties of the current axes.

`reset(gcf)` resets the properties of the current figure.

See Also `cla`, `clf`, `gca`, `gcf`, `hold`

Purpose Reshape array

Syntax

```
B = reshape(A,m,n)
B = reshape(A,m,n,p,...)
B = reshape(A,[m n p ...])
B = reshape(A,...,[],...)
B = reshape(A,siz)
```

Description

`B = reshape(A,m,n)` returns the `m`-by-`n` matrix `B` whose elements are taken column-wise from `A`. An error results if `A` does not have `m*n` elements.

`B = reshape(A,m,n,p,...)` or `B = reshape(A,[m n p ...])` returns an N-D array with the same elements as `A` but reshaped to have the size `m`-by-`n`-by-`p`-by-... . The product of the specified dimensions, `m*n*p*...`, must be the same as `prod(size(A))`.

`B = reshape(A,...,[],...)` calculates the length of the dimension represented by the placeholder `[]`, such that the product of the dimensions equals `prod(size(A))`. The value of `prod(size(A))` must be evenly divisible by the product of the specified dimensions. You can use only one occurence of `[]`.

`B = reshape(A,siz)` returns an N-D array with the same elements as `A`, but reshaped to `siz`, a vector representing the dimensions of the reshaped array. The quantity `prod(siz)` must be the same as `prod(size(A))`.

Examples

Reshape a 3-by-4 matrix into a 2-by-6 matrix.

```
A =
    1    4    7    10
    2    5    8    11
    3    6    9    12

B = reshape(A,2,6)

B =
    1    3    5    7    9   11
    2    4    6    8   10   12
B = reshape(A,2,[])
```

```
B =
     1     3     5     7     9    11
     2     4     6     8    10    12
```

See Also

`shiftdim`, `squeeze`

The colon operator `:`

Purpose

Convert between partial fraction expansion and polynomial coefficients

Syntax

```
[r,p,k] = residue(b,a)
[b,a] = residue(r,p,k)
```

Description

The `residue` function converts a quotient of polynomials to pole-residue representation, and back again.

`[r,p,k] = residue(b,a)` finds the residues, poles, and direct term of a partial fraction expansion of the ratio of two polynomials, $b(s)$ and $a(s)$, of the form

$$\frac{b(s)}{a(s)} = \frac{b_1 s^m + b_2 s^{m-1} + b_3 s^{m-2} + \ldots + b_{m+1}}{a_1 s^n + a_2 s^{n-1} + a_3 s^{n-2} + \ldots + a_{n+1}}$$

where b_j and a_j are the `j`th elements of the input vectors `b` and `a`.

`[b,a] = residue(r,p,k)` converts the partial fraction expansion back to the polynomials with coefficients in `b` and `a`.

Definition

If there are no multiple roots, then

$$\frac{b(s)}{a(s)} = \frac{r_1}{s - p_1} + \frac{r_2}{s - p_2} + \ldots + \frac{r_n}{s - p_n} + k(s)$$

The number of poles `n` is

```
n = length(a)-1 = length(r) = length(p)
```

The direct term coefficient vector is empty if `length(b) < length(a)`; otherwise

```
length(k) = length(b)-length(a)+1
```

If `p(j) = ... = p(j+m-1)` is a pole of multiplicity `m`, then the expansion includes terms of the form

$$\frac{r_j}{s - p_j} + \frac{r_{j+1}}{(s - p_j)^2} + \ldots + \frac{r_{j+m-1}}{(s - p_j)^m}$$

Arguments

`b,a`	Vectors that specify the coefficients of the polynomials in descending powers of s
`r`	Column vector of residues
`p`	Column vector of poles
`k`	Row vector of direct terms

Algorithm

It first obtains the poles with `roots`. Next, if the fraction is nonproper, the direct term `k` is found using `deconv`, which performs polynomial long division. Finally, the residues are determined by evaluating the polynomial with individual roots removed. For repeated roots, `resi2` computes the residues at the repeated root locations.

Limitations

Numerically, the partial fraction expansion of a ratio of polynomials represents an ill-posed problem. If the denominator polynomial, $a(s)$, is near a polynomial with multiple roots, then small changes in the data, including roundoff errors, can make arbitrarily large changes in the resulting poles and residues. Problem formulations making use of state-space or zero-pole representations are preferable.

Examples

If the ratio of two polynomials is expressed as

$$\frac{b(s)}{a(s)} = \frac{5s^3 + 3s^2 - 2s + 7}{-4s^3 + 8s + 3}$$

then

```
b = [ 5 3 -2 7]
a = [-4 0 8 3]
```

and you can calculate the partial fraction expansion as

```
[r, p, k] = residue(b,a)

r =
   -1.4167
   -0.6653
    1.3320
```

```
p =
    1.5737
   -1.1644
   -0.4093

k =
   -1.2500
```

Now, convert the partial fraction expansion back to polynomial coefficients.

```
[b,a] = residue(r,p,k)

b =
   -1.2500   -0.7500    0.5000   -1.7500

a =
    1.0000   -0.0000   -2.0000   -0.7500
```

The result can be expressed as

$$\frac{b(s)}{a(s)} = \frac{-1.25s^3 - 0.75s^2 + 0.50s - 1.75}{s^3 - 2.00s - 0.75}$$

Note that the result is normalized for the leading coefficient in the denominator.

See Also

`deconv`, `poly`, `roots`

References

[1] Oppenheim, A.V. and R.W. Schafer, *Digital Signal Processing*, Prentice-Hall, 1975, p. 56.

return

Purpose Return to the invoking function

Syntax

```
return
```

Description `return` causes a normal return to the invoking function or to the keyboard. It also terminates keyboard mode.

Examples If the determinant function were an M-file, it might use a `return` statement in handling the special case of an empty matrix as follows:

```
function d = det(A)
%DET det(A) is the determinant of A.
if isempty(A)
    d = 1;
    return
else
    ...
end
```

See Also `break`, `disp`, `end`, `error`, `for`, `if`, `keyboard`, `switch`, `while`

Purpose Convert RGB colormap to HSV colormap

Syntax

```
cmap = rgb2hsv(M)
```

Description

`cmap = rgb2hsv(M)` converts an RGB colormap, `M`, to an HSV colormap, `cmap`. Both colormaps are *m*-by-3 matrices. The elements of both colormaps are in the range 0 to 1.

The columns of the input matrix, `M`, represent intensities of red, green, and blue, respectively. The columns of the output matrix, `cmap`, represent hue, saturation, and value, respectively.

`hsv_image = rgb2hsv(rgb_image)` converts the RGB image (3-D array) to the equivalent HSV image (3-D array).

See Also

`brighten`, `colormap`, `hsv2rgb`,`rgbplot`

rgbplot

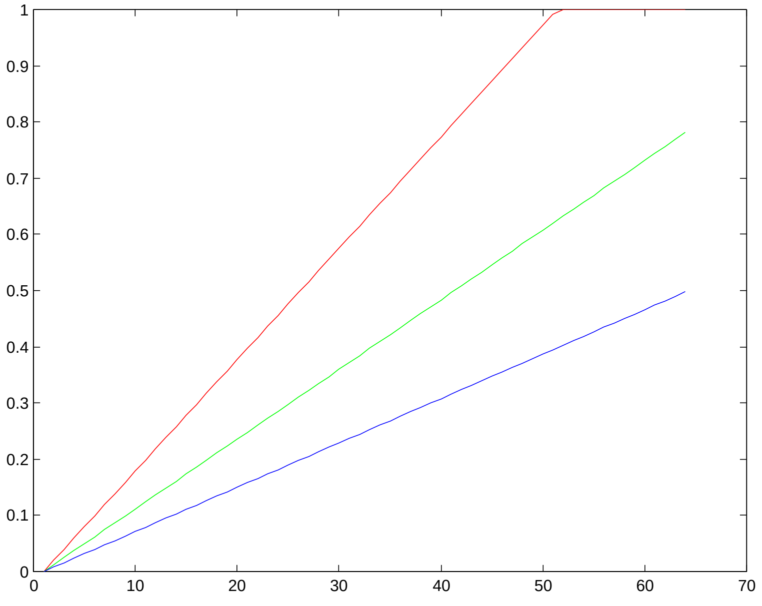
Purpose Plot colormap

Syntax `rgbplot(cmap)`

Description `rgbplot(cmap)` plots the three columns of `cmap`, where `cmap` is an *m*-by-3 colormap matrix. `rgbplot` draws the first column in red, the second in green, and the third in blue.

Examples Plot the RGB values of the copper colormap.

```
rgbplot(copper)
```



See Also `colormap`

Purpose Ribbon plot

Syntax

```
ribbon(Y)
ribbon(X,Y)
ribbon(X,Y,width)
h = ribbon(...)
```

Description

`ribbon(Y)` plots the columns of `Y` as separate three-dimensional ribbons using `X = 1:size(Y,1)`.

`ribbon(X,Y)` plots `X` versus the columns of `Y` as three-dimensional strips. `X` and `Y` are vectors of the same size or matrices of the same size. Additionally, `X` can be a row or a column vector, and `Y` a matrix with `length(X)` rows.

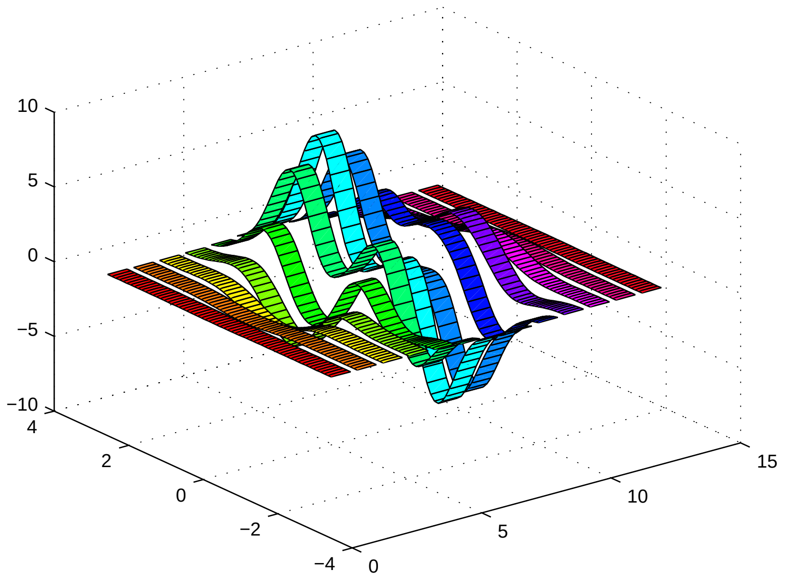
`ribbon(X,Y,width)` specifies the width of the ribbons. The default is `0.75`.

`h = ribbon(...)` returns a vector of handles to surface graphics objects. `ribbon` returns one handle per strip.

Examples

Create a ribbon plot of the peaks function.

```
[x,y] = meshgrid(-3:.5:3,-3:.1:3);
z = peaks(x,y);
ribbon(y,z)
colormap hsv
```



See Also

`plot`, `plot3`, `surface`, `waterfall`

Purpose

Remove application-defined data

Syntax

```
rmappdata(h,name,value)
```

Description

`rmappdata(h,name,value)` removes the application-defined data `name` from the object specified by handle `h`.

See Also

`getappdata`, `isappdata`, `setappdata`

rmfield

Purpose Remove structure fields

Syntax

```
s = rmfield(s,'field')
s = rmfield(s,FIELDS)
```

Description `s = rmfield(s,'field')` removes the specified field from the structure array `s`.

`s = rmfield(s,FIELDS)` removes more than one field at a time when `FIELDS` is a character array of field names or cell array of strings.

See Also `getfield`, `setfield`, `fieldnames`

Purpose Remove directories from MATLAB search path

Graphical Interface As an alternative to the `rmpath` function, use the **Set Path** dialog box.To open it, select **Set Path** from the **File** menu in the MATLAB desktop.

Syntax

```
rmpath('directory')
rmpath directory
```

Description `rmpath('directory')` removes the specified directory from MATLAB's current search path. Use the full pathname for `directory`.

`rmpath directory` is the unquoted form of the syntax.

Examples To remove `/usr/local/matlab/mytools` from the search path, type

```
rmpath /usr/local/matlab/mytools
```

See Also `addpath`, `path`, `rehash`, `pathtool`

root object

Purpose

Root object properties

Description

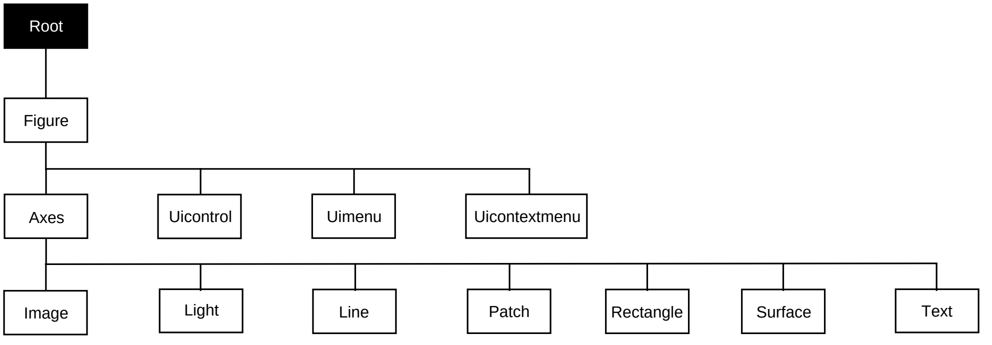
The root is a graphics object that corresponds to the computer screen. There is only one root object and it has no parent. The children of the root object are figures.

The root object exists when you start MATLAB; you never have to create it and you cannot destroy it. Use `set` and `get` to access the root properties.

See Also

`diary`, `echo`, `figure`, `format`, `gcf`, `get`, `set`

Object Hierarchy



Property List

The following table lists all root properties and provides a brief description of each. The property name links take you to an expanded description of the properties. This table does not include properties that are defined for, but not used by, the root object.

Property Name	Property Description	Property Value
Information about MATLAB's state		
`CallbackObject`	Handle of object whose callback is executing	Values: object handle
`CurrentFigure`	Handle of current figure	Values: object handle
`ErrorMessage`	Text of last error message	Value: character string
`PointerLocation`	Current location of pointer	Values: *x*-, and *y*-coordinates
`PointerWindow`	Handle of window containing the pointer	Values: figure handle
`ShowHiddenHandles`	Show or hide handles marked as hidden	Values: `on`, `off` Default: `off`
Controlling MATLAB's behavior		
`Diary`	Enable the diary file	Values: `on`, `off` Default: `off`
`DiaryFile`	Name of the diary file	Values: filename (string) Default: `diary`
`Echo`	Display each line of script M-file as executed	Values: `on`, `off` Default: `off`
`Format`	Format used to display numbers	Values: `short`, `shortE`, `long`, `longE`, `bank`, `hex`, `+`, `rat` Default: `shortE`
`FormatSpacing`	Display or omit extra line feed	Values: `compact`, `loose` Default: `loose`
`Language`	System environment setting	Values: string Default: `english`

Property Name	Property Description	Property Value
RecursionLimit	Maximum number of nested M-file calls	Values: integer Defalut: 2.1478e+009
Units	Units for PointerLocation and ScreenSize properties	Values: pixels, normalized, inches, centimeters, points, characters Default: pixels
Information about the display		
FixedWidthFontName	Value for axes, text, and uicontrol FontName property	Values: font name Default: Courier
ScreenDepth	Depth of the display bitmap	Values: bits per pixel
ScreenSize	Size of the screen	Values: [left, bottom, width, height]
General Information About Root Objects		
Children	Handles of all nonhidden Figue objects	Values: vector of handles
Parent	The root object has no parent	Value: [] (empty matrix)
Selected	This property is not used by the root object.	
Tag	User-specified label	Value: any string Default: '' (empty string)
Type	The type of graphics object (read only)	Value: the string 'root'
UserData	User-specified data	Values: any matrix Default: [] (empty matrix)

Root Properties

This section lists property names along with the type of values each accepts. Curly braces { } enclose default values.

`BusyAction` `cancel | {queue}`

Not used by the root object.

`ButtonDownFcn` string

Not used by the root object.

`CallbackObject` handle (read only)

Handle of current callback's object. This property contains the handle of the object whose callback routine is currently executing. If no callback routines are executing, this property contains the empty matrix [ ]. See also the `gco` command.

`CaptureMatrix` (obsolete)

This property has been superseded by the `getframe` command.

`CaptureRect` (obsolete)

This property has been superseded by the `getframe` command.

`Children` vector of handles

Handles of child objects. A vector containing the handles of all nonhidden figure objects. You can change the order of the handles and thereby change the stacking order of the figures on the display.

`Clipping` `{on} | off`

Clipping has no effect on the root object.

`CreateFcn`

The root does not use this property.

`CurrentFigure` figure handle

Handle of the current figure window, which is the one most recently created, clicked in, or made current with the statement:

```
figure(h)
```

which restacks the figure to the top of the screen, or

```
set(0,'CurrentFigure',h)
```

which does not restack the figures. In these statements, `h` is the handle of an existing figure. If there are no figure objects,

```
get(0,'CurrentFigure')
```

returns the empty matrix. Note, however, that `gcf` always returns a figure handle, and creates one if there are no figure objects.

DeleteFcn string

This property is not used since you cannot delete the root object

Diary `on | {off}`

Diary file mode. When this property is `on`, MATLAB maintains a file (whose name is specified by the `DiaryFile` property) that saves a copy of all keyboard input and most of the resulting output. See also the `diary` command.

DiaryFile string

Diary filename. The name of the diary file. The default name is `diary`.

Echo `on | {off}`

Script echoing mode. When `Echo` is `on`, MATLAB displays each line of a script file as it executes. See also the `echo` command.

ErrorMessage string

Text of last error message. This property contains the last error message issued by MATLAB.

FixedWidthFontName `font name`

Fixed-width font to use for axes, text, and uicontrols whose `FontName` *is set to* `FixedWidth`. MATLAB uses the font name specified for this property as the value for axes, text, and uicontrol `FontName` properties when their `FontName` property is set to `FixedWidth`. Specifying the font name with this property eliminates the need to hardcode font names in MATLAB applications and thereby enables these applications to run without modification in locales where non-ASCII character sets are required. In these cases, MATLAB attempts to set the value of `FixedWidthFontName` to the correct value for a given locale.

MATLAB application developers should not change this property, but should create axes, text, and uicontrols with `FontName` properties set to `FixedWidth` when they want to use a fixed width font for these objects.

MATLAB end users can set this property if they do not want to use the preselected value. In locales where Latin-based characters are used, Courier is the default.

`Format` `short | {shortE} | long | longE | bank | hex | + | rat`

Output format mode. This property sets the format used to display numbers. See also the `format` command.

- `short` – Fixed-point format with 5 digits.
- `shortE` – Floating-point format with 5 digits.
- `shortG` – Fixed- or floating-point format displaying as many significant figures as possible with 5 digits.
- `long` – Scaled fixed-point format with 15 digits.
- `longE` – Floating-point format with 15 digits.
- `longG` – Fixed- or floating-point format displaying as many significant figures as possible with 15 digits.
- `bank` – Fixed-format of dollars and cents.
- `hex` – Hexadecimal format.
- `+` – Displays + and – symbols.
- `rat` – Approximation by ratio of small integers.

`FormatSpacing` `compact | {loose}`

Output format spacing (see also `format` command).

- `compact` — Suppress extra line feeds for more compact display.
- `loose` — Display extra line feeds for a more readable display.

`HandleVisibility` `{on} | callback | off`

This property is not useful on the root object.

`HitTest` `{on} | off`

This property is not useful on the root object.

`Interruptible` `{on} | off`

This property is not useful on the root object.

`Language` string

System environment setting.

`Parent` handle

Handle of parent object. This property always contains the empty matrix, as the root object has no parent.

`PointerLocation` `[x,y]`

Current location of pointer. A vector containing the *x*- and *y*-coordinates of the pointer position, measured from the lower-left corner of the screen. You can move the pointer by changing the values of this property. The `Units` property determines the units of this measurement.

This property always contains the instantaneous pointer location, even if the pointer is not in a MATLAB window. A callback routine querying the `PointerLocation` can get a different value than the location of the pointer when the callback was triggered. This difference results from delays in callback execution caused by competition for system resources.

`PointerWindow` handle (read only)

Handle of window containing the pointer. MATLAB sets this property to the handle of the figure window containing the pointer. If the pointer is not in a MATLAB window, the value of this property is 0. A callback routine querying the `PointerWindow` can get the wrong window handle if you move the pointer to another window before the callback executes. This error results from delays in callback execution caused by competition for system resources.

`RecursionLimit` integer

Number of nested M-file calls. This property sets a limit to the number of nested calls to M-files MATLAB will make before stoping (or potentially running out of memory). By default the value is set to a large value. Setting this property to a smaller value (something like 150, for example) should prevent MATLAB from running out of memory and will instead cause MATLAB to issue an error when the limit is reached.

`ScreenDepth` bits per pixel

Screen depth. The depth of the display bitmap (i.e., the number of bits per pixel). The maximum number of simultaneously displayed colors on the current graphics device is 2 raised to this power.

`ScreenDepth` supersedes the `BlackAndWhite` property. To override automatic hardware checking, set this property to `1`. This value causes MATLAB to assume the display is monochrome. This is useful if MATLAB is running on color hardware but is displaying on a monochrome terminal. Such a situation can cause MATLAB to determine erroneously that the display is color.

`ScreenSize` 4-element rectangle vector (read only)

Screen size. A four-element vector,

```
[left,bottom,width,height]
```

that defines the display size. `left` and `bottom` are 0 for all `Units` except `pixels`, in which case `left` and `bottom` are 1. `width` and `height` are the screen dimensions in units specified by the `Units` property.

`Selected` `on | off`

This property has no effect on the root level.

`SelectionHighlight` `{on} | off`

This property has no effect on the root level.

`ShowHiddenHandles` `on | {off}`

Show or hide handles marked as hidden. When set to `on`, this property disables handle hiding and exposes all object handles, regardless of the setting of an object's `HandleVisibility` property. When set to `off`, all objects so marked remain hidden within the graphics hierarchy.

`Tag` string

User-specified object label. The `Tag` property provides a means to identify graphics objects with a user-specified label. While it is not necessary to identify the root object with a tag (since its handle is always `0`), you can use this property to store any string value that you can later retrieve using `set`.

`Type` string (read only)

Class of graphics object. For the root object, `Type` is always `'root'`.

`UIContextMenu` handle

This property has no effect on the root level.

Units `{pixels} | normalized | inches | centimeters | points | characters`

Unit of measurement. This property specifies the units MATLAB uses to interpret size and location data. All units are measured from the lower-left corner of the screen. Normalized units map the lower-left corner of the screen to (0,0) and the upper right corner to (1.0,1.0). `inches`, `centimeters`, and `points` are absolute units (one point equals 1/72 of an inch). Characters are units defined by characters from the default system font; the width of one unit is the width of the letter x, the height of one character is the distance between the baselines of two lines of text.

This property affects the `PointerLocation` and `ScreenSize` properties. If you change the value of `Units`, it is good practice to return it to its default value after completing your operation so as not to affect other functions that assume `Units` is set to the default value.

UserData matrix

User specified data. This property can be any data you want to associate with the root object. MATLAB does not use this property, but you can access it using the `set` and `get` functions.

Visible `{on} | off`

Object visibility. This property has no effect on the root object.

Purpose

Polynomial roots

Syntax

```
r = roots(c)
```

Description

`r = roots(c)` returns a column vector whose elements are the roots of the polynomial `c`.

Row vector `c` contains the coefficients of a polynomial, ordered in descending powers. If `c` has `n+1` components, the polynomial it represents is $c_1 s^n + \ldots + c_n s + c_{n+1}$.

Remarks

Note the relationship of this function to `p = poly(r)`, which returns a row vector whose elements are the coefficients of the polynomial. For vectors, `roots` and `poly` are inverse functions of each other, up to ordering, scaling, and roundoff error.

Examples

The polynomial $s^3 - 6s^2 - 72s - 27$ is represented in MATLAB as

```
p = [1 -6 -72 -27]
```

The roots of this polynomial are returned in a column vector by

```
r = roots(p)
r =
    12.1229
    -5.7345
    -0.3884
```

Algorithm

The algorithm simply involves computing the eigenvalues of the companion matrix:

```
A = diag(ones(n-2,1),-1);
A(1,:) = -c(2:n-1)./c(1);
eig(A)
```

It is possible to prove that the results produced are the exact eigenvalues of a matrix within roundoff error of the companion matrix `A`, but this does not mean that they are the exact roots of a polynomial with coefficients within roundoff error of those in `c`.

See Also

`fzero`, `poly`, `residue`

rose

Purpose Angle histogram

Syntax

```
rose(theta)
rose(theta,x)
rose(theta,nbins)
[tout,rout] = rose(...)
```

Description

`rose` creates an angle histogram, which is a polar plot showing the distribution of values grouped according to their numeric range. Each group is shown as one bin.

`rose(theta)` plots an angle histogram showing the distribution of `theta` in 20 angle bins or less. The vector `theta`, expressed in radians, determines the angle from the origin of each bin. The length of each bin reflects the number of elements in `theta` that fall within a group, which ranges from 0 to the greatest number of elements deposited in any one bin.

`rose(theta,x)` uses the vector `x` to specify the number and the locations of bins. `length(x)` is the number of bins and the values of `x` specify the center angle of each bin. For example, if `x` is a five-element vector, `rose` distributes the elements of `theta` in five bins centered at the specified `x` values.

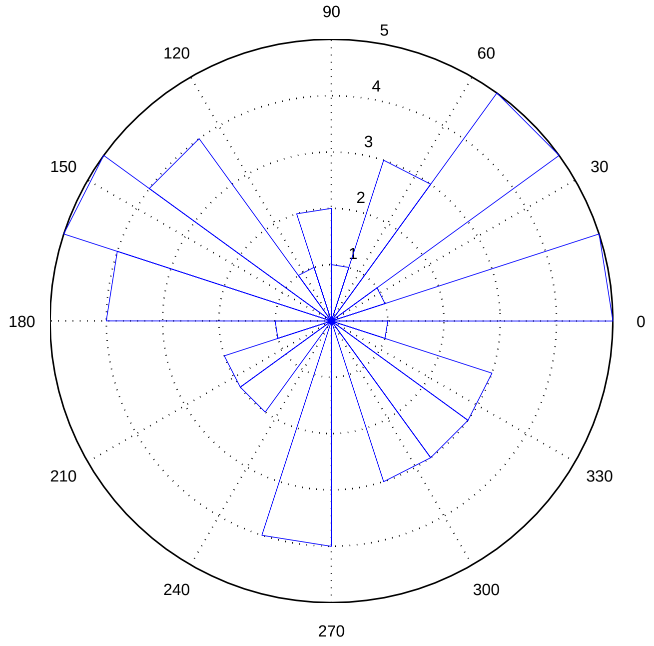
`rose(theta,nbins)` plots `nbins` equally spaced bins in the range `[0, 2*pi]`. The default is `20`.

`[tout,rout] = rose(...)` returns the vectors `tout` and `rout` so `polar(tout,rout)` generates the histogram for the data. This syntax does not generate a plot.

Example

Create a rose plot showing the distribution of 50 random numbers.

```
theta = 2*pi*rand(1,50);
rose(theta)
```



See Also compass, feather, hist, polar

Purpose Classic symmetric eigenvalue test problem

Syntax

```
A = rosser
```

Description

`A = rosser` returns the Rosser matrix. This matrix was a challenge for many matrix eigenvalue algorithms. But LAPACK's `DSYEV` routine used in MATLAB has no trouble with it. The matrix is 8-by-8 with integer elements. It has:

- A double eigenvalue
- Three nearly equal eigenvalues
- Dominant eigenvalues of opposite sign
- A zero eigenvalue
- A small, nonzero eigenvalue

Examples

```
rosser

ans =

   611   196  -192   407    -8   -52   -49    29
   196   899   113  -192   -71   -43    -8   -44
  -192   113   899   196    61    49     8    52
   407  -192   196   611     8    44    59   -23
    -8   -71    61     8   411  -599   208   208
   -52   -43    49    44  -599   411   208   208
   -49    -8     8    59   208   208    99  -911
    29   -44    52   -23   208   208  -911    99
```

Purpose

Rotate matrix 90°

Syntax

```
B = rot90(A)
B = rot90(A,k)
```

Description

`B = rot90(A)` rotates matrix `A` counterclockwise by 90 degrees.

`B = rot90(A,k)` rotates matrix `A` counterclockwise by `k*90` degrees, where `k` is an integer.

Examples

The matrix

```
X =
     1     2     3
     4     5     6
     7     8     9
```

rotated by 90 degrees is

```
Y = rot90(X)
Y =
     3     6     9
     2     5     8
     1     4     7
```

See Also

`flipdim`, `fliplr`, `flipud`

Purpose Rotate object about a specified direction

Syntax

```
rotate(h,direction,alpha)
rotate(...,origin)
```

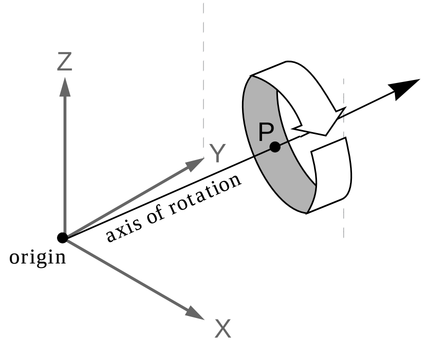
Description The `rotate` function rotates a graphics object in three-dimensional space, according to the right-hand rule.

`rotate(h,direction,alpha)` rotates the graphics object `h` by `alpha` degrees. `direction` is a two- or three-element vector that describes the axis of rotation in conjunction with the origin.

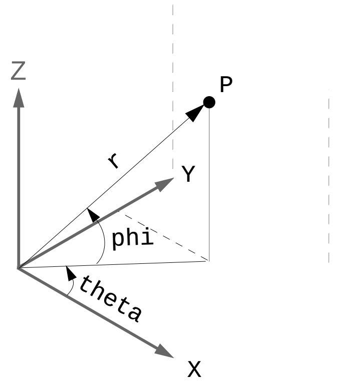
`rotate(...,origin)` specifies the origin of the axis of rotation as a three-element vector. The default origin is the center of the plot box.

Remarks The graphics object you want rotated must be a child of the same axes. The object's data is modified by the rotation transformation. This is in contrast to `view` and `rotate3d`, which only modify the viewpoint.

The axis of rotation is defined by an origin and a point *P* relative to the origin. *P* is expressed as the spherical coordinates `[theta phi]`, or as Cartesian coordinates.



The two-element form for `direction` specifies the axis direction using the spherical coordinates `[theta phi]`. `theta` is the angle in the *xy* plane counterclockwise from the positive *x*-axis. `phi` is the elevation of the direction vector from the *xy* plane.



The three-element form for `direction` specifies the axis direction using Cartesian coordinates. The direction vector is the vector from the origin to (X,Y,Z).

Examples

Rotate a graphics object 180° about the *x*-axis.

```
h = surf(peaks(20));
rotate(h,[1 0 0],180)
```

Rotate a surface graphics object 45° about its center in the *z* direction.

```
h = surf(peaks(20));
zdir = [0 0 1];
center = [10 10 0];
rotate(h,zdir,45,center)
```

Remarks

`rotate` changes the `Xdata`, `Ydata`, and `Zdata` properties of the appropriate graphics object.

See Also

`rotate3d`, `sph2cart`, `view`

The axes `CameraPosition`, `CameraTarget`, `CameraUpVector`, `CameraViewAngle`

rotate3d

Purpose Rotate axes using mouse

Syntax

```
rotate3d
rotate3d on
rotate3d off
```

Description `rotate3d on` enables interactive axes rotation within the current figure using the mouse. When interactive axes rotation is enabled, clicking on an axes draws an animated box, which rotates as the mouse is dragged, showing the view that will result when the mouse button is released. A numeric readout appears in the lower-left corner of the figure during this time, showing the current azimuth and elevation of the animated box. Releasing the mouse button removes the animated box and the readout, and changes the view of the axes to correspond to the last orientation of the animated box.

`rotate3d off` disables interactive axes rotation in the current figure.

`rotate3d` toggles interactive axes rotation in the current figure.

Double clicking on the figure restores the original view.

See Also `camorbit`, `rotate`, `view`

round

Purpose Round to nearest integer

Syntax `Y = round(X)`

Description `Y = round(X)` rounds the elements of `X` to the nearest integers. For complex `X`, the imaginary and real parts are rounded independently.

Examples

```
a = [-1.9, -0.2, 3.4, 5.6, 7.0, 2.4+3.6i]

a =
  Columns 1 through 4
  -1.9000          -0.2000            3.4000             5.6000
  Columns 5 through 6
   7.0000             2.4000 + 3.6000i

round(a)

ans =
  Columns 1 through 4
  -2.0000                0            3.0000             6.0000
  Columns 5 through 6
   7.0000             2.0000 + 4.0000i
```

See Also `ceil`, `fix`, `floor`

rref

Purpose

Reduced row echelon form

Syntax

```
R = rref(A)
[R,jb] = rref(A)
[R,jb] = rref(A,tol)
```

Description

`R = rref(A)` produces the reduced row echelon form of `A` using Gauss Jordan elimination with partial pivoting. A default tolerance of (`max(size(A))*eps *norm(A,inf)`) tests for negligible column elements.

`[R,jb] = rref(A)` also returns a vector `jb` so that:

- `r = length(jb)` is this algorithm's idea of the rank of `A`,
- `x(jb)` are the bound variables in a linear system `Ax = b`,
- `A(:,jb)` is a basis for the range of `A`,
- `R(1:r,jb)` is the `r`-by-`r` identity matrix.

`[R,jb] = rref(A,tol)` uses the given tolerance in the rank tests.

Roundoff errors may cause this algorithm to compute a different value for the rank than `rank`, `orth` and `null`.

Note The demo `rrefmovie(A)` shows a movie of the algorithm working.

Examples

Use `rref` on a rank-deficient magic square:

```
A = magic(4), R = rref(A)
A =
    16     2     3    13
     5    11    10     8
     9     7     6    12
     4    14    15     1
R =
     1     0     0     1
     0     1     0     3
     0     0     1    -3
     0     0     0     0
```

See Also inv, lu, rank

rsf2csf

Purpose

Convert real Schur form to complex Schur form

Syntax

```
[U,T] = rsf2csf(U,T)
```

Description

The *complex Schur form* of a matrix is upper triangular with the eigenvalues of the matrix on the diagonal. The *real Schur form* has the real eigenvalues on the diagonal and the complex eigenvalues in 2-by-2 blocks on the diagonal.

`[U,T] = rsf2csf(U,T)` converts the real Schur form to the complex form.

Arguments `U` and `T` represent the unitary and Schur forms of a matrix `A`, respectively, that satisfy the relationships: `A = U*T*U'` and `U'*U = eye(size(A))`. See `schur` for details.

Examples

Given matrix `A`,

```
 1     1     1     3
 1     2     1     1
 1     1     3     1
-2     1     1     4
```

with the eigenvalues

```
4.8121    1.9202 + 1.4742i    1.9202 + 1.4742i    1.3474
```

Generating the Schur form of `A` and converting to the complex Schur form

```
[u,t] = schur(A);
[U,T] = rsf2csf(u,t)
```

yields a triangular matrix `T` whose diagonal (underlined here for readability) consists of the eigenvalues of `A`.

```
U =

-0.4916       -0.2756 - 0.4411i    0.2133 + 0.5699i   -0.3428
-0.4980       -0.1012 + 0.2163i   -0.1046 + 0.2093i    0.8001
-0.6751        0.1842 + 0.3860i   -0.1867 - 0.3808i   -0.4260
-0.2337        0.2635 - 0.6481i    0.3134 - 0.5448i    0.2466
```

```
T =

4.8121        -0.9697 + 1.0778i   -0.5212 + 2.0051i   -1.0067
     0         1.9202 + 1.4742i    2.3355              0.1117 + 1.6547i
     0              0              1.9202 - 1.4742i    0.8002 + 0.2310i
     0              0                   0              1.3474
```

See Also

schur

run

Purpose Run a script

Syntax

```
run scriptname
```

Description

`run scriptname` runs the MATLAB script specified by `scriptname`. If `scriptname` contains the full pathname to the script file, then `run` changes the current directory to be the one in which the script file resides, executes the script, and sets the current directory back to what it was. The script is run within the caller's workspace.

`run` is a convenience function that runs scripts that are not currently on the path. Typically, you just type the name of a script at the MATLAB prompt to execute it. This works when the script is on your path. Use the `cd` or `addpath` function to make a script executable by entering the script name alone.

See Also `cd`, `addpath`

Purpose

Emulate the runtime environment in MATLAB and set the global error mode

Syntax

```
runtime on
runtime off
runtime status
runtime errormode mode
```

Description

The `runtime` command lets you emulate the Runtime Server environment in commercial MATLAB and set the global error mode for a runtime application. Because the Runtime Server disables the command window, it is generally much more convenient to test and debug with MATLAB emulating the Runtime Server than with the Runtime Server variant itself.

`runtime on` tells commercial MATLAB to begin emulating the Runtime Server. This means that MATLAB executes neither M-files nor standard P-files. The command line remains accessible.

`runtime off` returns MATLAB to its ordinary state.

`runtime status` indicates whether MATLAB is emulating the Runtime Server or not.

`runtime errormode` *`mode`* sets the global error mode to *`mode`*. The value of *`mode`* can be either `continue`, `quit`, or `dialog`. However, `dialog` is both the default error mode and the recommended one.

The error mode setting is only effective when the application runs with the Runtime Server; when the application runs with commercial MATLAB emulating the Runtime Server, untrapped errors are always displayed in the command window.

See Also

`isruntime`

Purpose

Save workspace variables on disk

Graphical Interface

As an alternative to the `save` function, select **Save Workspace As** from the **File** menu in the MATLAB desktop, or use the Workspace browser.

Syntax

```
save
save filename
save filename var1 var2 ...
save ... option
save('filename', ...)
```

Description

`save` by itself, stores all workspace variables in a binary format in the current directory in a file named `matlab.mat`. Retrieve the data with `load`. MAT-files are double-precision, binary, MATLAB format files. They can be created on one machine and later read by MATLAB on another machine with a different floating-point format, retaining as much accuracy and range as the different formats allow. They can also be manipulated by other programs external to MATLAB.

`save filename` stores all workspace variables in the current directory in `filename.mat`. To save to another directory, use the full pathname for the `filename`. If `filename` is the special string `stdio`, the `save` command sends the data as standard output.

`save filename var1 var2 ...` saves only the specified workspace variables in `filename.mat`. Use the `*` wildcard to save only those variables that match the specified pattern. For example, `save('A*')` saves all variables that start with `A`.

`save ... option` saves the workspace variables in the format specified by *`option`*

option Argument	Result: How Data is Stored
`-append`	The specified existed MAT-file, appended to the end
`-ascii`	8-digit ASCII format

option Argument	Result: How Data is Stored
`-ascii -double`	16-digit ASCII format
`-ascii -tabs`	delimits with tabs
`-ascii -double -tabs`	16-digit ASCII format, tab delimited
`-mat`	Binary MAT-file form (default)
`-v4`	A format that MATLAB version 4 can open

Remarks

When saving in ASCII format, consider the following:

- Each variable to be saved must be either a two dimensional double array or a two dimensional character array. Saving a complex double array causes the imaginary part of the data to be lost, as MATLAB cannot load nonnumeric data (`'i'`).
- In order to be able to read the file with the MATLAB `load` function, all of the variables must have the same number of columns. If you are using a program other than MATLAB to read the saved data this restriction can be relaxed.
- Each MATLAB character in a character array is converted to a floating point number equal to its internal ASCII code and written out as a floating point number string. There is no information in the save file that indicates whether the value was originally a number or a character.
- The values of all variables saved merge into a single variable that takes the name of the ASCII file (minus any extension). Therefore, it is advisable to save only one variable at a time.

With the `v4` flag, you can only save data constructs that are compatible with versions of MATLAB 4. Therefore, you cannot save structures, cell arrays, multidimensional arrays, or objects. In addition, you must use filenames that are supported by MATLAB version 4.

`save('filename', ...)` is the function form of the syntax.

For more control over the format of the file, MATLAB provides other functions, as listed in “See Also”, below.

Algorithm

The binary formats used by save depend on the size and type of each array. Arrays with any noninteger entries and arrays with 10,000 or fewer elements are saved in floating-point formats requiring 8 bytes per real element. Arrays with all integer entries and more than 10,000 elements are saved in the formats shown, requiring fewer bytes per element.

Element Range	Bytes per Element
0 to 255	1
0 to 65535	2
-32767 to 32767	2
$-2^{31}+1$ to $2^{31}-1$	4
other	8

External Interfaces to MATLAB provides details on reading and writing MAT-files from external C or Fortran programs. It is important to use recommended access methods, rather than rely upon the specific MAT-file format, which is likely to change in the future.

Examples

To save all variables from the workspace in binary MAT-file, `test.mat`, type

```
save test.mat
```

To save variables `p` and `q` in binary MAT-file, `test.mat`, type

```
savefile = 'test.mat';
p = rand(1,10);
q = ones(10);
save(savefile,'p','q')
```

To save the variables `vol` and `temp` in ASCII format to a file named `june10`, type

```
save('d:\mymfiles\june10','vol','temp','-ASCII’)
```

See Also

`diary`, `fprintf`, `fwrite`, `load`, `workspace`

Purpose Serialize an ActiveX control object to a file.

Syntax

```
save(h, filename)
```

Arguments

`h`
A MATLAB ActiveX object.

`filename`
The full pathname of the serialized data.

Description Save the ActiveX control object associated with the interface represented by the MATLAB ActiveX object `H` into a file. `filename` is the full pathname of the serialized data.

Example

```
h = actxcontrol('MwSamp.mwsampctrl.1');
   save(h, 'c:\temp\mycontrol.acx');
```

save (serial)

Purpose Save serial port objects and variables to a MAT-file

Syntax

```
save filename
save filename obj1 obj2...
```

Arguments

`filename`	The MAT-file name.
`obj1 obj2...`	Serial port objects or arrays of serial port objects.

Description

`save filename` saves all MATLAB variables to the MAT-file `filename`. If an extension is not specified for `filename`, then the `.mat` extension is used.

`save filename obj1 obj2...` saves the serial port objects `obj1` `obj2` ... to the MAT-file `filename`.

Remarks

You can use `save` in the functional form as well as the command form shown above. When using the functional form, you must specify the filename and serial port objects as strings. For example. to save the serial port object `s` to the file `MySerial.mat`

```
s = serial('COM1');
save('MySerial','s')
```

Any data that is associated with the serial port object is not automatically stored in the MAT-file. For example, suppose there is data in the input buffer for `obj`. To save that data to a MAT-file, you must bring it into the MATLAB workspace using one of the synchronous read functions, and then save to the MAT-file using a separate variable name. You can also save data to a text file with the `record` function.

You return objects and variables to the MATLAB workspace with the `load` command. Values for read-only properties are restored to their default values upon loading. For example, the `Status` property is restored to `closed`. To determine if a property is read-only, examine its reference pages.

If you use the `help` command to display help for `save`, then you need to supply the pathname shown below.

```
help serial/private/save
```

Example This example illustrates how to use the command and functional form of `save`.

```
s = serial('COM1');
set(s,'BaudRate',2400,'StopBits',1)
save MySerial1 s
set(s,'BytesAvailableFcn',@mycallback)
save('MySerial2','s')
```

See Also

Functions

`load`, `record`

Properties

`Status`

saveas

Purpose Save figure or model using specified format

Syntax

```
saveas(h,'filename.ext')
saveas(h,'filename','format')
```

Description `saveas(h,'filename.ext')` saves the figure or model with the handle `h` to the file `filename.ext`. The format of the file is determined by the extension, `ext`. Allowable values for `ext` are listed in this table.

ext Values	Format
`ai`	Adobe Illustrator '88
`bmp`	Windows bitmap
`emf`	Enhanced metafile
`eps`	EPS Level 1
`fig`	MATLAB figure (invalid for MATLAB models)
`jpg`	JPEG image (invalid for MATLAB models)
`m`	MATLAB M-file (invalid for MATLAB models)
`pbm`	Portable bitmap
`pcx`	Paintbrush 24-bit
`pgm`	Portable Graymap
`png`	Portable Network Graphics
`ppm`	Portable Pixmap
`tif`	TIFF image, compressed

`saveas(h,'filename','format')` saves the figure or model with the handle `h` to the file called `filename` using the specified `format`. The filename can have an extension but the extension is not used to define the file format. If no extension is specified, the standard extension corresponding to the specified format is automatically appended to the filename.

Allowable values for format are the extensions in the table above and the device types supported by print. The print device types include the formats listed in the table of extensions above as well as additional file formats. Use an extension from the table above or from the list of device types supported by print. When using the print device type to specify format for saveas, do not use the prepended -d.

Remarks

You can use open to open files saved using saveas with an m or fig extension. Other formats are not supported by open. The **Save As** dialog box you access from the figure window's **File** menu uses saveas, limiting the file extensions to m and fig. The **Export** dialog box you access from the figure window's **File** menu uses saveas with the format argument.

Examples

Example 1 – Specify File Extension

Save the current figure that you annotated using the Plot Editor to a file named pred_prey using the MATLAB fig format. This allows you to open the file pred_prey.fig at a later time and continue editing it with the Plot Editor.

```
saveas(gcf,'pred_prey.fig')
```

Example 2 – Specify File Format but No Extension

Save the current figure, using Adobe Illustrator format, to the file logo. Use the ai extension from the above table to specify the format. The file created is logo.ai.

```
saveas(gcf,'logo', 'ai')
```

This is the same as using the Adobe Illustrator format from the print devices table, which is -dill; use doc print or help print to see the table for print device types. The file created is logo.ai. MATLAB automatically appends the ai extension, for an Illustrator format file, because no extension was specified.

```
saveas(gcf,'logo', 'ill')
```

Example 3 – Specify File Format and Extension

Save the current figure to the file star.eps using the Level 2 Color PostScript format. If you use doc print or help print, you can see from the table for print device types that the device type for this format is -dpsc2. The file created is star.eps.

```
saveas(gcf,'star.eps', 'psc2')
```

In another example, save the current model to the file `trans.tiff` using the TIFF format with no compression. From the table for print device types, you can see the device type for this format is `-dtiffn`. The file created is `trans.tiff`.

```
saveas(gcf,'trans.tiff', 'tiffn')
```

See Also

`open`, `print`

Purpose Save an object to a MAT-file

Syntax

```
B = saveobj(A)
```

Description `B = saveobj(A)` is called by the MATLAB `save` function when object, `A`, is saved to a .MAT file. This call executes the `saveobj` method for the object's class, if such a method exists. The return value B is subsequently used by `save` to populate the .MAT file.

When you issue a `save` command on an object, MATLAB looks for a method called `saveobj` in the class directory. You can overload this method to modify the object before the save operation. For example, you could define a `saveobj` method that saves related data along with the object.

Remarks `saveobj` can be overloaded only for user objects. `save` will not call `saveobj` for a built-in datatype, such as `double`, even if `@double/saveobj` exists.

`saveobj` will be separately invoked for each object to be saved.

A child object does not inherit the `saveobj` method of its parent class. To implement `saveobj` for any class, including a class that inherits from a parent, you must define a `saveobj` method within that class directory.

Examples The following example shows a `saveobj` method written for the `portfolio` class. The method determines if a `portfolio` object has already been assigned an account number from a previous `save` operation. If not, `saveobj` calls `getAccountNumber` to obtain the number and assigns it to the `account_number` field. The contents of `b` is saved to the MAT-file.

```
function b = saveobj(a)
if isempty(a.account_number)
    a.account_number = getAccountNumber(a);
end
b = a;
```

See Also `save`, `load`, `loadobj`

Purpose 2-D Scatter plot

Syntax

```
scatter(X,Y,S,C)
scatter(X,Y)
scatter(X,Y,S)
scatter(...,markertype)
scatter(...,'filled')
h = scatter(...,)
```

Description

`scatter(X,Y,S,C)` displays colored circles at the locations specified by the vectors `X` and `Y` (which must be the same size).

`S` determines the area of each marker (specified in points^2). `S` can be a vector the same length as `X` and `Y` or a scalar. If `S` is a scalar, MATLAB draws all the markers the same size.

`C` determines the colors of each marker. When `C` is a vector the same length as `X` and `Y`, the values in `C` are linearly mapped to the colors in the current colormap. When `C` is a `length(X)`-by-3 matrix, it specifies the colors of the markers as RGB values. `C` can also be a color string (see `ColorSpec` for a list of color string specifiers)

`scatter(X,Y)` draws the markers in the default size and color.

`scatter(X,Y,S)` draws the markers at the specified sizes (`S`) with a single color.

`scatter(...,`*`markertype`*`)` uses the marker type specified instead of 'o' (see `LineSpec` for a list of marker specifiers).

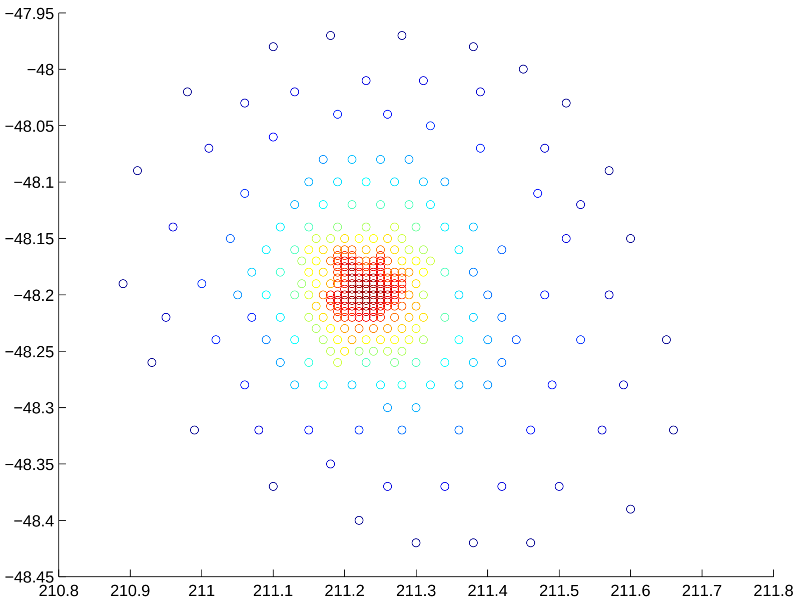
`scatter(...,'filled')` fills the markers.

`h = scatter(...)` returns the handles to the line objects created by `scatter` (see `line` for a list of properties you can specify using the object handles and `set`).

Remarks Use `plot` for single color, single marker size scatter plots.

Examples

```
load seamount
scatter(x,y,5,z)
```



See Also

`scatter3`, `plot`, `plotmatrix`

scatter3

Purpose

3-D scatter plot

Syntax

```
scatter3(X,Y,Z,S,C)
scatter3(X,Y,Z)
scatter3(X,Y,Z,S)
scatter3(...,markertype)
scatter3(...,'filled')
h = scatter3(...,)
```

Description

`scatter3(X,Y,Z,S,C)` displays colored circles at the locations specified by the vectors `X`, `Y`, and `Z` (which must all be the same size).

`S` determines the size of each marker (specified in points). `S` can be a vector the same length as `X`, `Y`, and `Z` or a scalar. If `S` is a scalar, MATLAB draws all the markers the same size.

`C` determines the colors of each marker. When `C` is a vector the same length as `X`, `Y`, and `Z`, the values in `C` are linearly mapped to the colors in the current colormap. When `C` is a `length(X)`-by-3 matrix, it specifies the colors of the markers as RGB values. `C` can also be a color string (see `ColorSpec` for a list of color string specifiers)

`scatter3(X,Y,Z)` draws the markers in the default size and color.

`scatter3(X,Y,Z,S)` draws the markers at the specified sizes (`S`) with a single color.

`scatter3(...,`*`markertype`*`)` uses the marker type specified instead of 'o' (see `LineSpec` for a list of marker specifiers).

`scatter3(...,'filled')` fills the markers.

`h = scatter3(...)` returns the handles to the line objects created by `scatter3` (see `line` for a list of properties you can specify using the object handles and `set`).

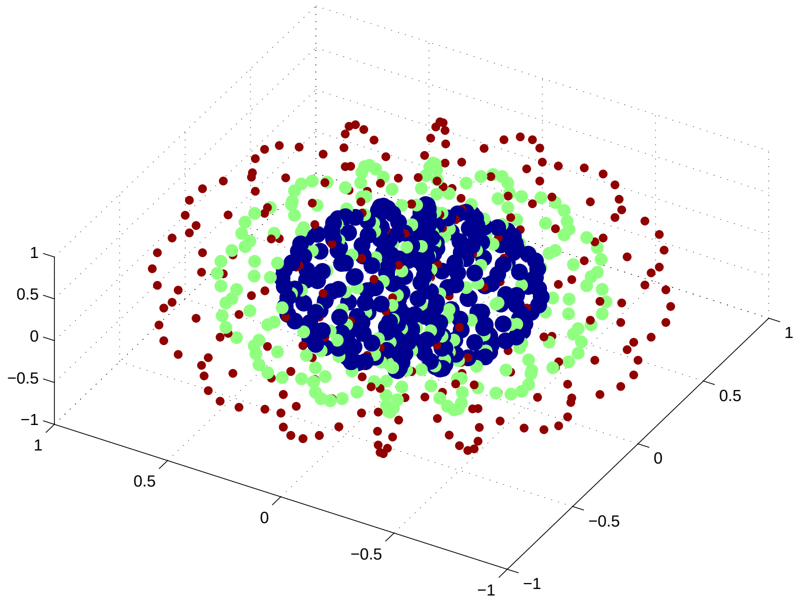
Remarks

Use `plot3` for single color, single marker size 3-D scatter plots.

Examples

```
[x,y,z] = sphere(16);
X = [x(:)*.5 x(:)*.75 x(:)];
Y = [y(:)*.5 y(:)*.75 y(:)];
```

```
Z = [z(:)*.5 z(:)*.75 z(:)];
S = repmat([1 .75 .5]*10,prod(size(x)),1);
C = repmat([1 2 3],prod(size(x)),1);
scatter3(X(:),Y(:),Z(:),S(:),C(:),'filled'), view(-60,60)
```



See Also `scatter`, `plot3`

schur

Purpose

Schur decomposition

Syntax

```
T = schur(A)
T = schur(A,flag)
[U,T] = schur(A,...)
```

Description

The `schur` command computes the Schur form of a matrix.

`T = schur(A)` returns the Schur matrix `T`.

`T = schur(A,flag)` for real matrix A, returns a Schur matrix `T` in one of two forms depending on the value of `flag`:

`'complex'`	`T` is triangular and is complex if `A` has complex eigenvalues.
`'real'`	`T` has the real eigenvalues on the diagonal and the complex eigenvalues in 2-by-2 blocks on the diagonal. `'real'` is the default.

If `A` is complex, `schur` returns the complex Schur form in matrix `T`. The complex Schur form is upper triangular with the eigenvalues of `A` on the diagonal.

The function `rsf2csf` converts the real Schur form to the complex Schur form.

`[U,T] = schur(A,...)` also returns a unitary matrix `U` so that `A = U*T*U'` and `U'*U = eye(size(A))`.

Examples

`H` is a 3-by-3 eigenvalue test matrix:

```
H = [ -149    -50   -154
       537    180    546
       -27     -9    -25 ]
```

Its Schur form is

```
schur(H)

ans =
    1.0000    7.1119   815.8706
         0    2.0000   -55.0236
         0         0     3.0000
```

The eigenvalues, which in this case are 1, 2, and 3, are on the diagonal. The fact that the off-diagonal elements are so large indicates that this matrix has poorly conditioned eigenvalues; small changes in the matrix elements produce relatively large changes in its eigenvalues.

Algorithm

schur uses LAPACK routines to compute the Schur form of a matrix:

Matrix A	Routine
Real symmetric	DSYTRD, DSTEQR DSYTRD, DORGTR, DSTEQR (with output U)
Real nonsymmetric	DGEHRD, DHSEQR DGEHRD, DORGHR, DHSEQR (with output U)
Complex Hermitian	ZHETRD, ZSTEQR ZHETRD, ZUNGTR, ZSTEQR (with output U)
Non-Hermitian	ZGEHRD, ZHSEQR ZGEHRD, ZUNGHR, ZHSEQR (with output U)

See Also

eig, hess, qz, rsf2csf

References

[1] Anderson, E., Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, *LAPACK User's Guide* (http://www.netlib.org/lapack/lug/lapack_lug.html), Third Edition, SIAM, Philadelphia, 1999.

script

Purpose Script M-files

Description A script file is an external file that contains a sequence of MATLAB statements. By typing the filename, subsequent MATLAB input is obtained from the file. Script files have a filename extension of `.m` and are often called M-files.

Scripts are the simplest kind of M-file. They are useful for automating blocks of MATLAB commands, such as computations you have to perform repeatedly from the command line. Scripts can operate on existing data in the workspace, or they can create new data on which to operate. Although scripts do not return output arguments, any variables that they create remain in the workspace so you can use them in further computations. In addition, scripts can produce graphical output using commands like `plot`.

Scripts can contain any series of MATLAB statements. They require no declarations or begin/end delimiters.

Like any M-file, scripts can contain comments. Any text following a percent sign (%) on a given line is comment text. Comments can appear on lines by themselves, or you can append them to the end of any executable line.

See Also `echo`, `function`, `type`

Purpose

Secant and hyperbolic secant

Syntax

```
Y = sec(X)
Y = sech(X)
```

Description

The sec and sech commands operate element-wise on arrays. The functions' domains and ranges include complex values. All angles are in radians.

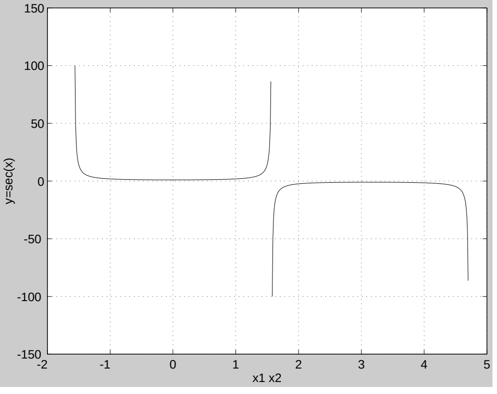
Y = sec(X) returns an array the same size as X containing the secant of the elements of X.

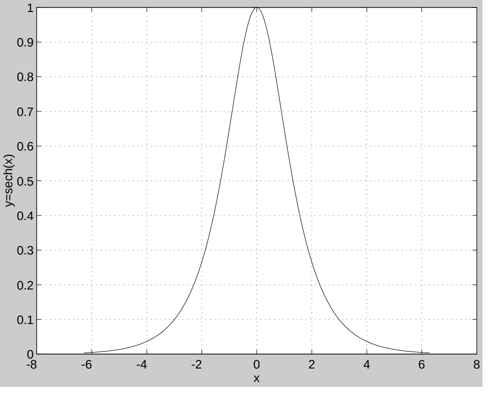
Y = sech(X) returns an array the same size as X containing the hyperbolic secant of the elements of X.

Examples

Graph the secant over the domains $-\pi/2 < x < \pi/2$ and $\pi/2 < x < 3\pi/2$, and the hyperbolic secant over the domain $-2\pi \le x \le 2\pi$.

```
x1 = -pi/2+0.01:0.01:pi/2-0.01;
x2 = pi/2+0.01:0.01:(3*pi/2)-0.01;
plot(x1,sec(x1),x2,sec(x2))
x = -2*pi:0.01:2*pi; plot(x,sech(x))
```





The expression `sec(pi/2)` does not evaluate as infinite but as the reciprocal of the floating-point accuracy `eps`, because `pi` is a floating-point approximation to the exact value of π.

Algorithm

`sec` and `sech` use these algorithms.

$$\sec(z) = \frac{1}{\cos(z)}$$

$$\operatorname{sech}(z) = \frac{1}{\cosh(z)}$$

See Also

asec, asech

Purpose Select, move, resize, or copy axes and uicontrol graphics objects

Syntax

```
A = selectmoveresize;
set(h,'ButtonDownFcn','selectmoveresize')
```

Description `selectmoveresize` is useful as the callback routine for axes and uicontrol button down functions. When executed, it selects the object and allows you to move, resize, and copy it.

For example, this statement sets the `ButtonDownFcn` of the current axes to `selectmoveresize`:

```
set(gca,'ButtonDownFcn','selectmoveresize')
```

`A = selectmoveresize` returns a structure array containing:

- `A.Type`: a string containing the action type, which can be `Select`, `Move`, `Resize`, or `Copy`.
- `A.Handles`: a list of the selected handles or for a `Copy` an m-by-2 matrix containing the original handles in the first column and the new handles in the second column.

See Also The `ButtonDownFcn` of axes and uicontrol graphics objects

Purpose

Semi-logarithmic plots

Syntax

```
semilogx(Y)
semilogx(X1,Y1,...)
semilogx(X1,Y1,LineSpec,...)
semilogx(...,'PropertyName',PropertyValue,...)
h = semilogx(...)

semilogy(...)
h = semilogy(...)
```

Description

`semilogx` and `semilogy` plot data as logarithmic scales for the *x*- and *y*-axis, respectively. logarithmic

`semilogx(Y)` creates a plot using a base 10 logarithmic scale for the *x*-axis and a linear scale for the *y*-axis. It plots the columns of `Y` versus their index if `Y` contains real numbers. `semilogx(Y)` is equivalent to `semilogx(real(Y), imag(Y))` if `Y` contains complex numbers. `semilogx` ignores the imaginary component in all other uses of this function.

`semilogx(X1,Y1,...)` plots all `Xn` versus `Yn` pairs. If only `Xn` or `Yn` is a matrix, `semilogx` plots the vector argument versus the rows or columns of the matrix, depending on whether the vector's row or column dimension matches the matrix.

`semilogx(X1,Y1,LineSpec,...)` plots all lines defined by the `Xn,Yn,LineSpec` triples. `LineSpec` determines line style, marker symbol, and color of the plotted lines.

`semilogx(...,'`*`PropertyName`*`',PropertyValue,...)` sets property values for all line graphics objects created by `semilogx`.

`semilogy(...)` creates a plot using a base 10 logarithmic scale for the *y*-axis and a linear scale for the *x*-axis.

`h = semilogx(...)` and `h = semilogy(...)` return a vector of handles to line graphics objects, one handle per line.

Remarks

If you do not specify a color when plotting more than one line, `semilogx` and `semilogy` automatically cycle through the colors and line styles in the order specified by the current axes `ColorOrder` and `LineStyleOrder` properties.

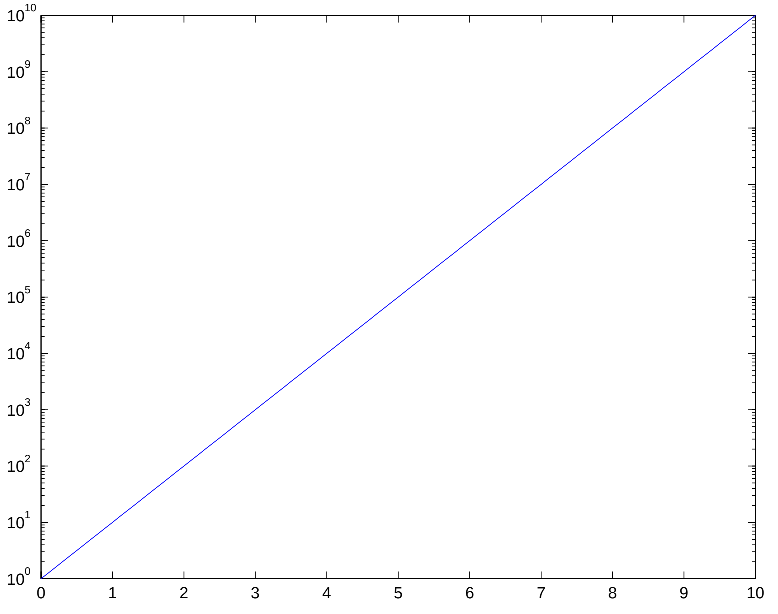
You can mix `Xn,Yn` pairs with `Xn,Yn,LineSpec` triples; for example,

```
semilogx(X1,Y1,X2,Y2,LineSpec,X3,Y3)
```

Examples

Create a simple `semilogy` plot.

```
x = 0:.1:10;
semilogy(x,10.^x)
```



See Also

`line`, `LineSpec`, `loglog`, `plot`

send (activex)

Purpose Returns a list of events that the control can trigger.

Syntax

```
send (a)
```

Arguments

`a`
Activex object returned by `actxcontrol`.

Description Displays a list of events that controls `send`.

Example

```
send (a)

Change = Void Change ()
Click = Void Click ()
DblClick = Void DblClick ()
KeyDown = Void KeyDown (Variant(Pointer), Short)
KeyPress = Void KeyPress (Variant(Pointer), Short)
KeyUp = Void KeyUp (Variant(Pointer), Short)
MouseDown = Void MouseDown (Short, Short, Vendor-Defined,
   Vendor-Defined)
MouseMove = Void MouseMove (Short, Short, Vendor-Defined,
   Vendor-Defined)
MouseUp = Void MouseUp (Short, Short, Vendor-Defined,
   Vendor-Defined)
```

Purpose

Create a serial port object

Syntax

```
obj = serial('port')
obj = serial('port','PropertyName',PropertyValue,...)
```

Arguments

`'port'`	The serial port name.
`'PropertyName'`	A serial port property name.
`PropertyValue`	A property value supported by *PropertyName*.
`obj`	The serial port object.

Description

`obj = serial('port')` creates a serial port object associated with the serial port specified by `port`. If `port` does not exist, or if it is in use, you will not be able to connect the serial port object to the device.

`obj = serial('port','PropertyName',PropertyValue,...)` creates a serial port object with the specified property names and property values. If an invalid property name or property value is specified, an error is returned and the serial port object is not created.

Remarks

When you create a serial port object, these property values are automatically configured:

- The `Type` property is given by `serial`.
- The `Name` property is given by concatenating `Serial` with the port specified in the `serial` function.
- The `Port` property is given by the port specified in the `serial` function.

You can specify the property names and property values using any format supported by the `set` function. For example, you can use property name/property value cell array pairs. Additionally, you can specify property names without regard to case, and you can make use of property name completion. For example, the following commands are all valid.

```
s = serial('COM1','BaudRate',4800);
s = serial('COM1','baudrate',4800);
s = serial('COM1','BAUD',4800);
```

Refer to "Configuring Property Values" for a list of serial port object properties that you can use with `serial`.

Before you can communicate with the device, it must be connected to `obj` with the `fopen` function. A connected serial port object has a `Status` property value of `open`. An error is returned if you attempt a read or write operation while the object is not connected to the device. You can connect only one serial port object to a given serial port.

Example

This example creates the serial port object `s1` associated with the serial port COM1.

```
s1 = serial('COM1');
```

The `Type`, `Name`, and `Port` properties are automatically configured.

```
get(s1,{'Type','Name','Port'})
ans =
    'serial'    'Serial-COM1'    'COM1'
```

To specify properties during object creation

```
s2 = serial('COM2','BaudRate',1200,'DataBits',7);
```

See Also

Functions

`fclose`, `fopen`

Properties

`Name`, `Port`, `Status`, `Type`

Purpose

Send a break to the device connected to the serial port

Syntax

```
serialbreak(obj)
serialbreak(obj,time)
```

Arguments

`obj`	A serial port object.
`time`	The duration of the break, in milliseconds.

Description

`serialbreak(obj)` sends a break of 10 milliseconds to the device connected to `obj`.

`serialbreak(obj,time)` sends a break to the device with a duration, in milliseconds, specified by `time`. Note that the duration of the break may be inaccurate under some operating systems.

Remarks

For some devices, the break signal provides a way to clear the hardware buffer.

Before you can send a break to the device, it must be connected to `obj` with the `fopen` function. A connected serial port object has a `Status` property value of `open`. An error is returned if you attempt to send a break while `obj` is not connected to the device.

`serialbreak` is a synchronous function, and blocks the command line until execution is complete.

If you issue `serialbreak` while data is being asynchronously written, an error is returned. In this case, you must call the `stopasync` function or wait for the write operation to complete.

See Also

Functions

`fopen`, `stopasync`

Properties

`Status`

set

Purpose Set object properties

Syntax

```
set(H,'PropertyName',PropertyValue,...)
set(H,a)
set(H,pn,pv...)
set(H,pn,<m-by-n cell array>)
a= set(h)
a= set(0,'Factory')
a= set(0,'FactoryObjectTypePropertyName')
a= set(h,'Default')
a= set(h,'DefaultObjectTypePropertyName')
<cell array> = set(h,'PropertyName')
```

Description

`set(H,'PropertyName',PropertyValue,...)` sets the named properties to the specified values on the object(s) identified by `H`. `H` can be a vector of handles, in which case `set` sets the properties' values for all the objects.

`set(H,a)` sets the named properties to the specified values on the object(s) identified by `H`. `a` is a structure array whose field names are the object property names and whose field values are the values of the corresponding properties.

`set(H,pn,pv,...)` sets the named properties specified in the cell array `pn` to the corresponding value in the cell array `pv` for all objects identified in `H`.

`set(H,pn,<m-by-n cell array>)` sets `n` property values on each of `m` graphics objects, where `m = length(H)` and `n` is equal to the number of property names contained in the cell array `pn`. This allows you to set a given group of properties to different values on each object.

`a = set(h)` returns the user-settable properties and possible values for the object identified by `h`. `a` is a structure array whose field names are the object's property names and whose field values are the possible values of the corresponding properties. If you do not specify an output argument, MATLAB displays the information on the screen. `h` must be scalar.

`a = set(0,'Factory')` returns the properties whose defaults are user settable for all objects and lists possible values for each property. `a` is a structure array whose field names are the object's property names and whose field values are the possible values of the corresponding properties. If you do not specify an output argument, MATLAB displays the information on the screen.

`a = set(0,'Factory`*`ObjectTypePropertyName`*`')` returns the possible values of the named property for the specified object type, if the values are strings. The argument `Factory`*`ObjectTypePropertyName`* is the word `Factory` concatenated with the object type (e.g., `axes`) and the property name (e.g., `CameraPosition`).

`a = set(h,'Default')` returns the names of properties having default values set on the object identified by `h`. `set` also returns the possible values if they are strings. `h` must be scalar.

`a = set(h,'Default`*`ObjectTypePropertyName`*`')` returns the possible values of the named property for the specified object type, if the values are strings. The argument `Default`*`ObjectTypePropertyName`* is the word `Default` concatenated with the object type (e.g., `axes`) and the property name (e.g., `CameraPosition`). For example, `DefaultAxesCameraPosition`. `h` must be scalar.

`pv = set(h,'`*`PropertyName`*`')` returns the possible values for the named property. If the possible values are strings, `set` returns each in a cell of the cell array, `pv`. For other properties, `set` returns an empty cell array. If you do not specify an output argument, MATLAB displays the information on the screen. `h` must be scalar.

Remarks

You can use any combination of property name/property value pairs, structure arrays, and cell arrays in one call to `set`.

Examples

Set the `Color` property of the current axes to blue.

```
set(gca,'Color','b')
```

Change all the lines in a plot to black.

```
plot(peaks)
set(findobj('Type','line'),'Color','k')
```

You can define a group of properties in a structure to better organize your code. For example, these statements define a structure called `active`, which contains a set of property definitions used for the uicontrol objects in a particular figure. When this figure becomes the current figure, MATLAB changes colors and enables the controls.

```
active.BackgroundColor = [.7 .7 .7];
active.Enable = 'on';
active.ForegroundColor = [0 0 0];

if gcf == control_fig_handle
   set(findobj(control_fig_handle,'Type','uicontrol'),active)
end
```

You can use cell arrays to set properties to different values on each object. For example, these statements define a cell array to set three properties,

```
PropName(1) = {'BackgroundColor'};
PropName(2) = {'Enable'};
PropName(3) = {'ForegroundColor'};
```

These statements define a cell array containing three values for each of three objects (i.e., a 3-by-3 cell array).

```
PropVal(1,1) = {[.5 .5 .5]};
PropVal(1,2) = {'off'};
PropVal(1,3) = {[.9 .9 .9]};

PropVal(2,1) = {[1 0 0]};
PropVal(2,2) = {'on'};
PropVal(2,3) = {[1 1 1]};

PropVal(3,1) = {[.7 .7 .7]};
PropVal(3,2) = {'on'};
PropVal(3,3) = {[0 0 0]};
```

Now pass the arguments to `set`,

```
set(H,PropName,PropVal)
```

where `length(H) = 3` and each element is the handle to a uicontrol.

See Also

`findobj`, `gca`, `gcf`, `gco`, `gcbo`, `get`

Purpose Set an interface property to a specific value.

Syntax

```
set (a [, 'propertyname' [, value [, arg1, arg2, …]]])
```

Arguments

`a`
An `activex` object handle previously returned from `actxcontrol`, `actxserver`, `get`, or `invoke`.

`propertyname`
A string that is the name of the property to be set.

`value`
The value to which the interface property is set.

`arg1, …, argn`
Arguments, if any, required by the property. Properties are similar to methods in that it is possible for a property to have arguments.

Returns There is no return value from set.

Description Set an interface property to a specific value. See “Converting Data” in *MATLAB External Interfaces* for information on how MATLAB converts workspace matrices to ActiveX data types.

Example

```
f = figure ('pos', [100 200 200 200]);
% Create the control to fill the figure.
a = actxcontrol ('MWSAMP.MwsampCtrl.1', [0 0 200 200], f)
set (a, 'Label', 'Click to fire event');
set (a, 'Radius', 40);
invoke (a, 'Redraw');
```

set (serial)

Purpose

Configure or display serial port object properties

Syntax

```
set(obj)
props = set(obj)
set(obj,'PropertyName')
props = set(obj,'PropertyName')
set(obj,'PropertyName',PropertyValue,...)
set(obj,PN,PV)
set(obj,S)
```

Arguments

`obj`	A serial port object or an array of serial port objects.
`'PropertyName'`	A property name for `obj`.
`PropertyValue`	A property value supported by *`PropertyName`*.
`PN`	A cell array of property names.
`PV`	A cell array of property values.
`S`	A structure with property names and property values.
`props`	A structure array whose field names are the property names for `obj`, or cell array of possible values.

Description

`set(obj)` displays all configurable properties values for `obj`. If a property has a finite list of possible string values, then these values are also displayed.

`props = set(obj)` returns all configurable properties and their possible values for `obj` to `props`. `props` is a structure whose field names are the property names of `obj`, and whose values are cell arrays of possible property values. If the property does not have a finite set of possible values, then the cell array is empty.

`set(obj,'PropertyName')` displays the valid values for *`PropertyName`* if it possesses a finite list of string values.

`props = set(obj,'PropertyName')` returns the valid values for *`PropertyName`* to `props`. `props` is a cell array of possible string values or an empty cell array if *`PropertyName`* does not have a finite list of possible values.

`set(obj,'PropertyName',PropertyValue,...)` configures multiple property values with a single command.

`set(obj,PN,PV)` configures the properties specified in the cell array of strings `PN` to the corresponding values in the cell array `PV`. `PN` must be a vector. `PV` can be m-by-n where m is equal to the number of serial port objects in `obj` and n is equal to the length of `PN`.

`set(obj,S)` configures the named properties to the specified values for `obj`. `S` is a structure whose field names are serial port object properties, and whose field values are the values of the corresponding properties.

Remarks

Refer to "Configuring Property Values" for a list of serial port object properties that you can configure with `set`.

You can use any combination of property name/property value pairs, structures, and cell arrays in one call to `set`. Additionally, you can specify a property name without regard to case, and you can make use of property name completion. For example, if `s` is a serial port object, then the following commands are all valid.

```
set(s,'BaudRate')
set(s,'baudrate')
set(s,'BAUD')
```

If you use the `help` command to display help for `set`, then you need to supply the pathname shown below.

```
help serial/set
```

Examples

This example illustrates some of the ways you can use `set` to configure or return property values for the serial port object `s`.

```
s = serial('COM1');
set(s,'BaudRate',9600,'Parity','even')
set(s,{'StopBits','RecordName'},{2,'sydney.txt'})
set(s,'Parity')
[ {none} | odd | even | mark | space ]
```

See Also

Functions

`get`

setappdata

Purpose Set application-defined data

Syntax

```
setappdata(h,name,value)
```

Description `setappdata(h,name,value)` sets application-defined data for the object with handle `h`. The application-defined data, which is created if it does not already exist, is assigned a `name` and a `value`. `value` can be type of data.

See Also `getappdata`, `isappdata`, `rmappdata`

Purpose

Return the set difference of two vectors

Syntax

```
c = setdiff(A,B)
c = setdiff(A,B,'rows')
[c,i] = setdiff(...)
```

Description

`c = setdiff(A,B)` returns the values in A that are not in B. The resulting vector is sorted is ascending order. In set theoretic terms, `c = A - B`. A and B can be cell arrays of strings.

`c = setdiff(A,B,'rows')` when `A` and `B` are matrices with the same number of columns returns the rows from `A` that are not in `B`.

`[c,i] = setdiff(...)` also returns an index vector `index` such that `c = a(i)` or `c = a(i,:)`.

Examples

```
A = magic(5);
B = magic(4);
[c,i] = setdiff(A(:),B(:));
c' =    17    18    19    20    21    22    23    24    25
i' =     1    10    14    18    19    23     2     6    15
```

See Also

`intersect`, `ismember`, `setxor`, `union`, `unique`

setfield

Purpose

Set field of structure array

Syntax

```
s = setfield(s,'field',v)
s = setfield(s,{i,j},'field',{k},v)
```

Description

`s = setfield(s,'field',v)`, where `s` is a 1-by-1 structure, sets the contents of the specified field to the value `v`. This is equivalent to the syntax `s.field = v`.

`s = setfield(s,{i,j},'field',{k},v)` sets the contents of the specified field to the value `v`. This is equivalent to the syntax `s(i,j).field(k) = v`. All subscripts must be passed as cell arrays—that is, they must be enclosed in curly braces (similar to`{i,j}` and `{k}` above). Pass field references as strings.

Examples

Given the structure

```
mystr(1,1).name = 'alice';
mystr(1,1).ID = 0;
mystr(2,1).name = 'gertrude';
mystr(2,1).ID = 1;
```

You can change the `name` field of `mystr(2,1)` using

```
mystr = setfield(mystr,{2,1},'name','ted');
mystr(2,1).name

ans =

ted
```

The following example sets fields of a structure using `setfield` with variable and quoted field names and additional subscripting arguments.

```
class = 5;   student = 'John_Doe';
grades_Doe = [85,89,76,93,85,91,68,84,95,73];
grades = [];

grades = setfield(grades,{class}, student, 'Math',{10,21:30},...
                  grades_Doe);
```

You can check the outcome using the standard structure syntax.

```
grades(class).John_Doe.Math(10,21:30)

ans =

    85    89    76    93    85    91    68    84    95    73
```

See Also

getfield, rmfield, fieldnames

setstr

Purpose Set string flag

Description This MATLAB 4 function has been renamed `char` in MATLAB 5.

Purpose Set exclusive-or of two vectors

Syntax

```
c = setxor(A,B)
c = setxor(A,B,'rows')
[c,ia,ib] = setxor(...)
```

Description

`c = setxor(A,B)` returns the values that are not in the intersection of A and `B`. The resulting vector is sorted. A and `B` can be cell arrays of strings.

`c = setxor(A,B,'rows')` when `A` and `B` are matrices with the same number of columns returns the rows that are not in the intersection of `A` and `B`.

`[c,ia,ib] = setxor(...)` also returns index vectors `ia` and `ib` such that `c` is a sorted combination of the elements c = `a(ia)` and c = `b(ib)` or, for row combinations, `c = a(ia,:)` and `c = b(ib,:)`.

Examples

```
a = [-1 0 1 Inf -Inf NaN];
b = [-2 pi 0 Inf];
c = setxor(a,b)

c =
      -Inf   -2.0000   -1.0000    1.0000    3.1416       NaN
```

See Also

`intersect`, `ismember`, `setdiff`, `union`, `unique`

shading

Purpose

Set color shading properties

Syntax

```
shading flat
shading faceted
shading interp
```

Description

The `shading` function controls the color shading of surface and patch graphics objects.

`shading flat` each mesh line segment and face has a constant color determined by the color value at the end point of the segment or the corner of the face that has the smallest index or indices.

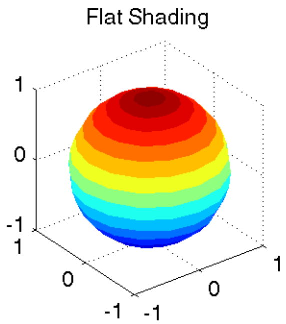
`shading faceted` flat shading with superimposed black mesh lines. This is the default shading mode.

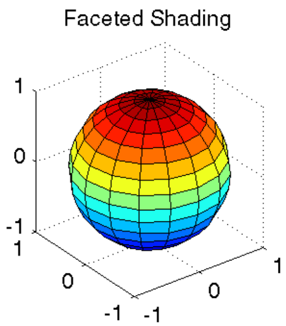
`shading interp` varies the color in each line segment and face by interpolating the colormap index or true color value across the line or face.

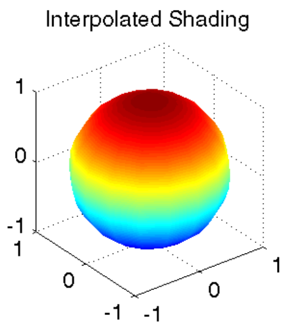
Examples

Compare a flat, faceted, and interpolated-shaded sphere.

```
subplot(3,1,1)
sphere(16)
axis square
shading flat
title('Flat Shading')

subplot(3,1,2)
sphere(16)
axis square
shading faceted
title('Faceted Shading')

subplot(3,1,3)
sphere(16)
axis square
shading interp
title('Interpolated Shading')
```







Algorithm

`shading` sets the `EdgeColor` and `FaceColor` properties of all surface and patch graphics objects in the current axes. `shading` sets the appropriate values, depending on whether the surface or patch objects represent meshes or solid surfaces.

See Also

`fill`, `fill3`, `hidden`, `mesh`, `patch`, `pcolor`, `surf`

The `EdgeColor` and `FaceColor` properties for surface and patch graphics objects.

Purpose Shift dimensions

Syntax

```
B = shiftdim(X,n)
[B,nshifts] = shiftdim(X)
```

Description `B = shiftdim(X,n)` shifts the dimensions of `X` by `n`. When `n` is positive, `shiftdim` shifts the dimensions to the left and wraps the `n` leading dimensions to the end. When `n` is negative, `shiftdim` shifts the dimensions to the right and pads with singletons.

`[B,nshifts] = shiftdim(X)` returns the array `B` with the same number of elements as `X` but with any leading singleton dimensions removed. A singleton dimension is any dimension for which `size(A,dim) = 1`. `nshifts` is the number of dimensions that are removed.

If `X` is a scalar, `shiftdim` has no effect.

Examples The `shiftdim` command is handy for creating functions that, like `sum` or `diff`, work along the first nonsingleton dimension.

```
a = rand(1,1,3,1,2);
[b,n]  = shiftdim(a); % b is 3-by-1-by-2 and n is 2.
c = shiftdim(b,-n);   % c == a.
d = shiftdim(a,3);    % d is 1-by-2-by-1-by-1-by-3.
```

See Also `reshape`, `squeeze`

shrinkfaces

Purpose

Reduce the size of patch faces

Syntax

```
shrinkfaces(p,sf)
nfv = shrinkfaces(p,sf)
nfv = shrinkfaces(fv,sf)
shrinkfaces(p), shrinkfaces(fv)
nfv = shrinkfaces(f,v,sf)
[nf,nv] = shrinkfaces(...)
```

Description

`shrinkfaces(p,sf)` shrinks the area of the faces in patch `p` to shrink factor `sf`. A shrink factor of 0.6 shrinks each face to 60% of its original area. If the patch contains shared vertices, MATLAB creates nonshared vertices before performing the face-area reduction.

`nfv = shrinkfaces(p,sf)` returns the face and vertex data in the struct `nfv`, but does not set the `Faces` and `Vertices` properties of patch `p`.

`nfv = shrinkfaces(fv,sf)` uses the face and vertex data from the struct `fv`.

`shrinkfaces(p)` and `shrinkfaces(fv)` (without specifying a shrink factor) assume a shrink factor of 0.3.

`nfv = shrinkfaces(f,v,sf)` uses the face and vertex data from the arrays `f` and `v`.

`[nf,nv] = shrinkfaces(...)` returns the face and vertex data in two separate arrays instead of a struct.

Examples

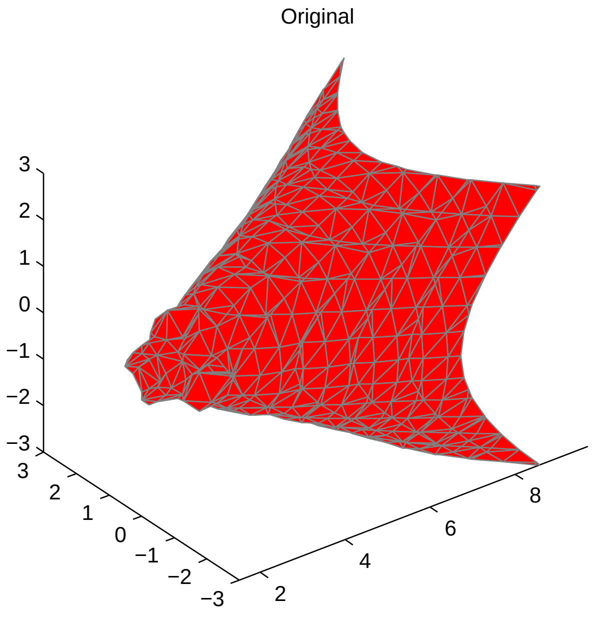
This example uses the flow data set, which represents the speed profile of a submerged jet within a infinite tank (type `help flow` for more information). Two isosurfaces provide a before and after view of the effects of shrinking the face size.

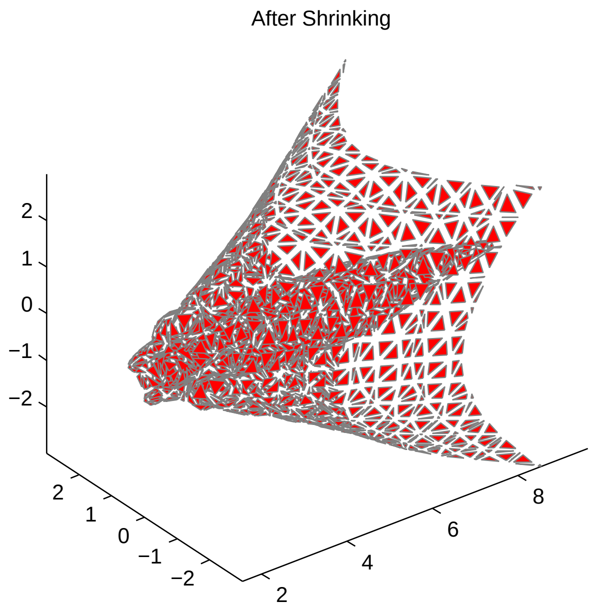
- First `reducevolume` samples the flow data at every other point and then `isosurface` generates the faces and vertices data.
- The `patch` command accepts the face/vertex struct and draws the first (`p1`) isosurface.
- Use the `daspect`, `view`, and `axis` commands to set up the view and then add a `title`.

- The shrinkfaces command modifies the face/vertex data and passes it directly to patch.

```
[x,y,z,v] = flow;
[x,y,z,v] = reducevolume(x,y,z,v,2);
fv = isosurface(x,y,z,v,-3);
p1 = patch(fv);
set(p1,'FaceColor','red','EdgeColor',[.5,.5,.5]);
daspect([1 1 1]); view(3); axis tight
title('Original')

figure
p2 = patch(shrinkfaces(fv,.3));
set(p2,'FaceColor','red','EdgeColor',[.5,.5,.5]);
daspect([1 1 1]); view(3); axis tight
```

```
title('After Shrinking')
```





See Also `isocaps`, `isonormals`, `isosurface`, `reducepatch`, `reducevolume`, `smooth3`, `subvolume`

sign

Purpose Signum function

Syntax

```
Y = sign(X)
```

Description

`Y = sign(X)` returns an array `Y` the same size as `X`, where each element of `Y` is:

- `1` if the corresponding element of `X` is greater than zero
- `0` if the corresponding element of `X` equals zero
- `-1` if the corresponding element of `X` is less than zero

For nonzero complex `X`, `sign(X) = X./abs(X)`.

See Also `abs`, `conj`, `imag`, `real`

Purpose

Sine and hyperbolic sine

Syntax

```
Y = sin(X)
Y = sinh(X)
```

Description

The `sin` and `sinh` commands operate element-wise on arrays. The functions' domains and ranges include complex values. All angles are in radians.

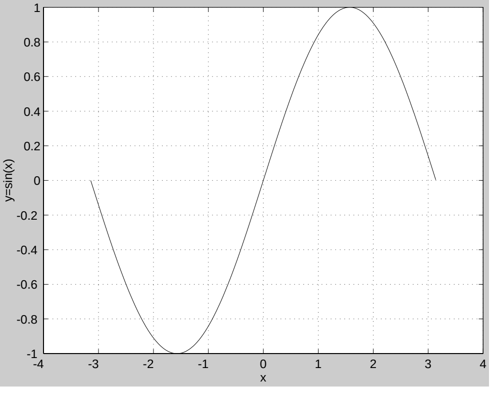
`Y = sin(X)` returns the circular sine of the elements of `X`.

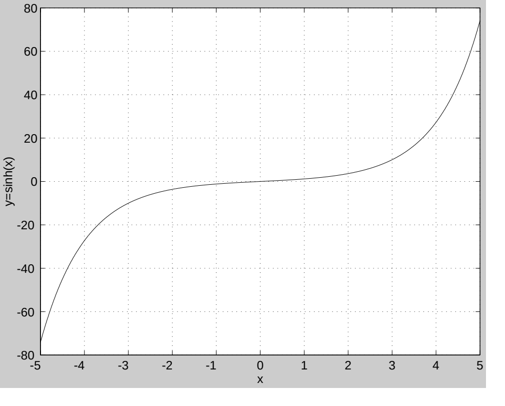
`Y = sinh(X)` returns the hyperbolic sine of the elements of `X`.

Examples

Graph the sine function over the domain $-\pi \le x \le \pi$, and the hyperbolic sine function over the domain $-5 \le x \le 5$.

```
x = -pi:0.01:pi; plot(x,sin(x))
x = -5:0.01:5; plot(x,sinh(x))
```





The expression `sin(pi)` is not exactly zero, but rather a value the size of the floating-point accuracy `eps`, because `pi` is only a floating-point approximation to the exact value of π.

Algorithm

`sin` and `sinh` use these algorithms.

$$\sin(x+iy) = \sin(x)\cos(y) + i\cos(x)\sin(y)$$

$$\sin(z) = \frac{e^{iz} - e^{-iz}}{2i}$$

$$\sinh(z) = \frac{e^{z} - e^{-z}}{2}$$

See Also

`asin`, `asinh`

Purpose Convert to single-precision

Syntax

```
B = single(A)
```

Description B = single(A) converts the matrix A to single-precision, returning that value in B. A can be any numeric object (such as a double). If A is already single-precision, single has no effect. Single-precision quantities require less storage than double-precision quantities, but have less precision and a smaller range.

The single class is primarily meant to be used to store single-precision values. Hence most operations that manipulate arrays without changing their elements are defined. Examples are reshape, size, the relational operators, subscripted assignment and subscripted reference. No math operations are defined for single objects.

You can define your own methods for the single class by placing the appropriately named method in an @single directory within a directory on your path.

Examples

```
a = magic(4);
b = single(a);

whos
  Name      Size         Bytes  Class

  a         4x4            128  double array
  b         4x4             64  single array
```

See Also double

size

Purpose

Array dimensions

Syntax

```
d = size(X)
[m,n] = size(X)
m = size(X,dim)
[d1,d2,d3,...,dn] = size(X)
```

Description

`d = size(X)` returns the sizes of each dimension of array `X` in a vector `d` with `ndims(X)` elements.

`[m,n] = size(X)` returns the size of matrix `X` in variables `m` and `n`.

`m = size(X,dim)` returns the size of the dimension of `X` specified by scalar `dim`.

`[d1,d2,d3,...,dn] = size(X)` returns the sizes of the various dimensions of array `X` in separate variables.

If the number of output arguments `n` does not equal `ndims(X)`, then

`If n > ndims(X)`	Ones are returned in the “extra” variables `dndims(X)+1` through `dn`.
`If n < ndims(X)`	The final variable `dn` contains the product of the sizes of all the “remaining” dimensions of `X`, that is, dimensions `n+1` through `ndims(X)`.

Examples

The size of the second dimension of `rand(2,3,4)` is 3.

```
m = size(rand(2,3,4),2)

m =
     3
```

Here the size is output as a single vector.

```
d = size(rand(2,3,4))

d =
     2     3     4
```

Here the size of each dimension is assigned to a separate variable.

```
[m,n,p] = size(rand(2,3,4))
m =
     2

n =
     3

p =
     4
```

If `X = ones(3,4,5)`, then

```
[d1,d2,d3] = size(X)

d1 =       d2 =       d3 =
     3          4          5
```

but when the number of output variables is less than `ndims(X)`:

```
[d1,d2] = size(X)

d1 =       d2 =
     3          20
```

The “extra” dimensions are collapsed into a single product.

If `n > ndims(X)`, the “extra” variables all represent singleton dimensions:

```
[d1,d2,d3,d4,d5,d6] = size(X)

d1 =       d2 =       d3 =
     3          4          5

d4 =       d5 =       d6 =
     1          1          1
```

See Also `exist`, `length`, `whos`

size (serial)

Purpose

Size of serial port object array

Syntax

```
d = size(obj)
[m,n] = size(obj)
[m1,m2,...,mn] = size(obj)
m = size(obj,dim)
```

Arguments

`obj`	A serial port object or an array of serial port objects.
`dim`	The dimension of `obj`.
`d`	The number of rows and columns in `obj`.
`m`	The number of rows in `obj`, or the length of the dimension specified by `dim`.
`n`	The number of columns in `obj`.
`m1,m2,...,mn`	The length of the first N dimensions of `obj`.

Description

`d = size(obj)` returns the two-element row vector `d` containing the number of rows and columns in `obj`.

`[m,n] = size(obj)` returns the number of rows and columns in separate output variables.

`[m1,m2,m3,...,mn] = size(obj)` returns the length of the first `n` dimensions of `obj`.

`m = size(obj,dim)` returns the length of the dimension specified by the scalar `dim`. For example, `size(obj,1)` returns the number of rows.

See Also

Functions

`length`

Purpose

Volumetric slice plot

Syntax

```
slice(V,sx,sy,sz)
slice(X,Y,Z,V,sx,sy,sz)
slice(V,XI,YI,ZI)
slice(X,Y,Z,V,XI,YI,ZI)
slice(...,'method')
h = slice(...)
```

Description

`slice` displays orthogonal slice planes through volumetric data.

`slice(V,sx,sy,sz)` draws slices along the x, y, z directions in the volume `V` at the points in the vectors `sx`, `sy`, and `sz`. `V` is an *m*-by-*n*-by-*p* volume array containing data values at the default location `X = 1:n`, `Y = 1:m`, `Z = 1:p`. Each element in the vectors `sx`, `sy`, and `sz` defines a slice plane in the *x*-, *y*-, or *z*-axis direction.

`slice(X,Y,Z,V,sx,sy,sz)` draws slices of the volume `V`. `X`, `Y`, and `Z` are three-dimensional arrays specifying the coordinates for `V`. `X`, `Y`, and `Z` must be monotonic and orthogonally spaced (as if produced by the function `meshgrid`). The color at each point is determined by 3-D interpolation into the volume `V`.

`slice(V,XI,YI,ZI)` draws data in the volume `V` for the slices defined by `XI`, `YI`, and `ZI`. `XI`, `YI`, and `ZI` are matrices that define a surface, and the volume is evaluated at the surface points. `XI`, `YI`, and `ZI` must all be the same size.

`slice(X,Y,Z,V,XI,YI,ZI)` draws slices through the volume `V` along the surface defined by the arrays `XI`, `YI`, `ZI`.

`slice(...,'method')` specifies the interpolation method. `'method'` is `'linear'`, `'cubic'`, or `'nearest'`.

- `linear` specifies trilinear interpolation (the default).
- `cubic` specifies tricubic interpolation.
- `nearest` specifies nearest neighbor interpolation.

`h = slice(...)` returns a vector of handles to surface graphics objects.

Remarks

The color drawn at each point is determined by interpolation into the volume V.

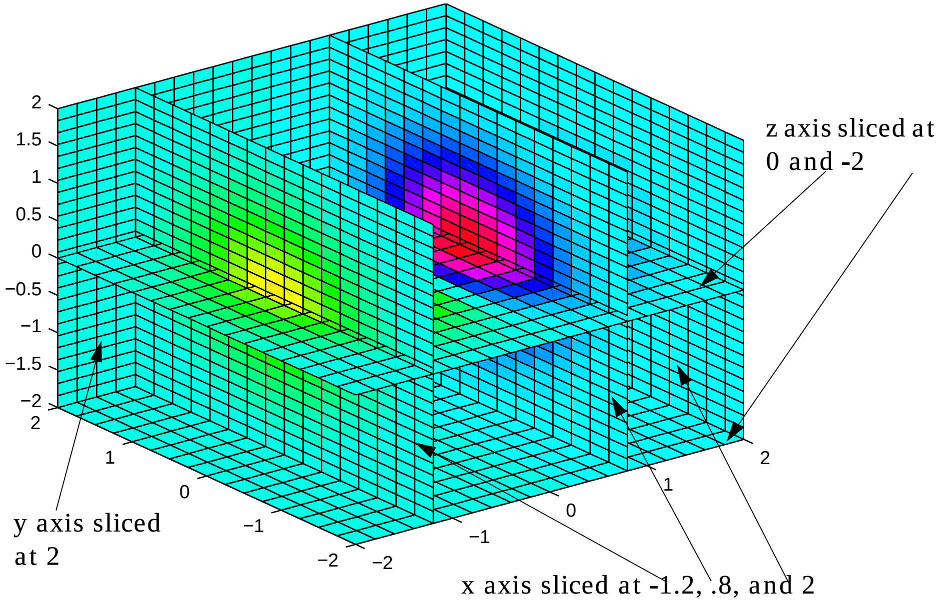
Examples

Visualize the function

$$v = xe^{(-x^2 - y^2 - z^2)}$$

over the range $-2 \leq x \leq 2$, $-2 \leq y \leq 2$, $-2 \leq z \leq 2$:

```
[x,y,z] = meshgrid(−2:.2:2,−2:.25:2,−2:.16:2);
v = x.*exp(−x.^2−y.^2−z.^2);
xslice = [−1.2,.8,2]; yslice = 2; zslice = [−2,0];
slice(x,y,z,v,xslice,yslice,zslice)
colormap hsv
```



Slicing At Arbitrary Angles

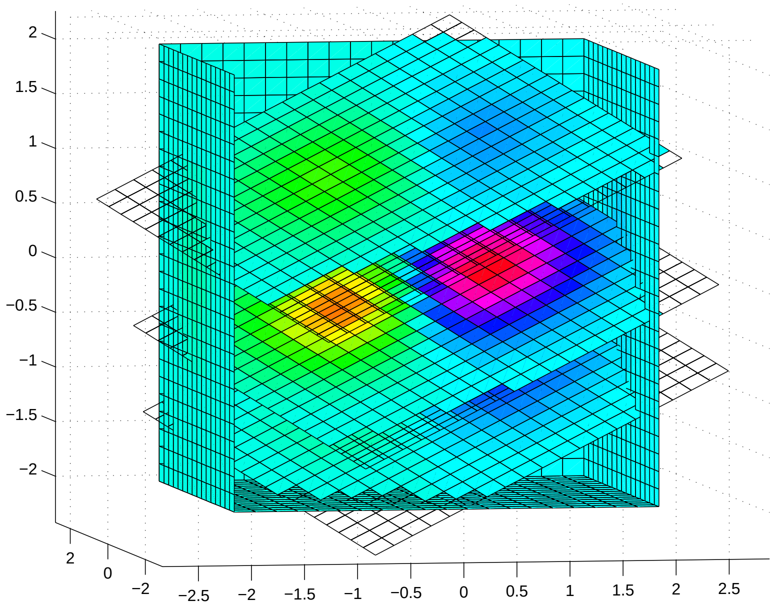
You can also create slices that are oriented in arbitrary planes. To do this,

- Create a slice surface in the domain of the volume (`surf`, `linspace`).
- Orient this surface with respect the the axes (`rotate`).
- Get the `XData`, `YData`, and `ZData` of the surface (`get`).
- Use this data to draw the slice plane within the volume.

For example, these statements slice the volume in the first example with a rotated plane. Placing these commands within a for loop “passes” the plane through the volume along the z-axis.

```
for i = –2:.5:2
   hsp = surf(linspace(–2,2,20),linspace(–2,2,20),zeros(20)+i);
   rotate(hsp,[1,–1,1],30)
   xd = get(hsp,'XData');
   yd = get(hsp,'YData');
   zd = get(hsp,'ZData');
   delete(hsp)
   slice(x,y,z,v,[–2,2],2,-2) % Draw some volume boundaries
   hold on
   slice(x,y,z,v,xd,yd,zd)
   hold off
   axis tight
   view(–5,10)
   drawnow
end
```

The following picture illustrates three positions of the same slice surface as it passes through the volume.



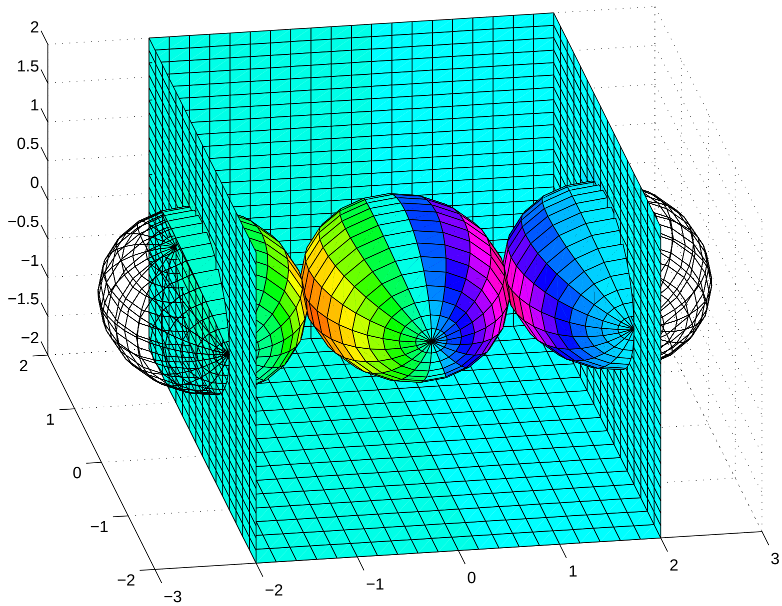
Slicing with a Nonplanar Surface

You can slice the volume with any surface. This example probes the volume created in the previous example by passing a spherical slice surface through the volume.

```
[xsp,ysp,zsp] = sphere;
slice(x,y,z,v,[-2,2],2,-2)  % Draw some volume boundaries

for i = -3:.2:3
   hsp = surface(xsp+i,ysp,zsp);
   rotate(hsp,[1 0 0],90)
   xd = get(hsp,'XData');
   yd = get(hsp,'YData');
   zd = get(hsp,'ZData');
   delete(hsp)
   hold on
   hslicer = slice(x,y,z,v,xd,yd,zd);
   axis tight
```

```
    xlim([-3,3])
    view(-10,35)
    drawnow
    delete(hslicer)
    hold off
end
```

The following picture illustrates three positions of the spherical slice surface as it passes through the volume.



See Also

`interp3`, `meshgrid`

smooth3

Purpose

Smooth 3-D data

Syntax

```
W = smooth3(V)
W = smooth3(V,'filter')
W = smooth3(V,'filter',size)
W = smooth3(V,'filter',size,sd)
```

Description

`W = smooth3(V)` smooths the input data `V` and returns the smoothed data in `W`.

`W = smooth3(V,'`*`filter`*`')` *`filter`* determines the convolution kernel and can be the strings `gaussian` or `box` (default).

`W = smooth3(V,'`*`filter`*`',size)` sets the size of the convolution kernel (default is `[3 3 3]`). If `size` is scalar, then `size` is interpreted as [`size`, `size`, `size`].

`W = smooth3(V,'`*`filter`*`',size,sd)` sets an attribute of the convolution kernel. When *`filter`* is `gaussian`, `sd` is the standard deviation (default is .65).

Examples

This example smooths some random 3-D data and then creates an isosurface with end caps.

```
data = rand(10,10,10);
data = smooth3(data,'box',5);
p1 = patch(isosurface(data,.5), ...
   'FaceColor','blue','EdgeColor','none');
p2 = patch(isocaps(data,.5), ...
    'FaceColor','interp','EdgeColor','none');
isonormals(data,p1)
view(3); axis vis3d tight
camlight; lighting phong
```

See Also

`isocaps`, `isonormals`, `isosurface`, `patch`, `reducepatch`, `reducevolume`, `subvolume`

Purpose

Sort elements in ascending order

Syntax

```
B = sort(A)
B = sort(A,dim)
[B,INDEX] = sort(A,...)
```

Description

`B = sort(A)` sorts the elements along different dimensions of an array, and arranges those elements in ascending order.

If A is a ...	sort(A) ...
Vector	Sorts the elements of `A` in ascending order.
Matrix	Sorts each column of `A` in ascending order.
Multidimensional array	Sorts `A` along the first non-singleton dimension, and returns an array of sorted vectors.
Cell array of strings	Sorts the strings in ASCII dictionary order.

Real, complex, and string elements are permitted. For elements of A with identical values, the order of these elements is preserved in the sorted list. When `A` is complex, the elements are sorted by magnitude, i.e., `abs(A)`, and where magnitudes are equal, further sorted by phase angle, i.e., `angle(A)`, on the interval $[-\pi, \pi]$. If `A` includes any `NaN` elements, `sort` places these at the end.

`B = sort(A,dim)` sorts the elements along the dimension of `A` specified by a scalar `dim`. If `dim` is a vector, `sort` works iteratively on the specified dimensions. Thus, `sort(A,[1 2])` is equivalent to `sort(sort(A,2),1)`.

`[B,IX] = sort(A,...)` also returns an array of indices `IX`, where `size(IX) == size(A)`. If `A` is a vector, `B = A(IX)`. If `A` is an `m`-by-`n` matrix, then each column of `IX` is a permutation vector of the corresponding column of `A`, such that

```
for j = 1:n
    B(:,j) = A(IX(:,j),j);
end
```

If A has repeated elements of equal value, the returned indices preserve the original ordering.

Examples

This example sorts a matrix A in each dimension, and then sorts it a third time, requesting an array of indices for the sorted result.

```
A = [ 3 7 5
      0 4 2 ];

sort(A,1)

ans =
     0     4     2
     3     7     5

sort(A,2)

ans =
     3     5     7
     0     2     4

[B,IX] = sort(A,2)

B =
     3     5     7
     0     2     4

IX =
     1     3     2
     1     3     2
```

See Also

max, mean, median, min, sortrows

Purpose Sort rows in ascending order

Syntax

```
B = sortrows(A)
B = sortrows(A,column)
[B,index] = sortrows(A)
```

Description `B = sortrows(A)` sorts the rows of `A` as a group in ascending order. Argument `A` must be either a matrix or a column vector.

For strings, this is the familiar dictionary sort. When `A` is complex, the elements are sorted by magnitude, and, where magnitudes are equal, further sorted by phase angle on the interval $[-\pi, \pi]$.

`B = sortrows(A,column)` sorts the matrix based on the columns specified in the vector `column`. For example, `sortrows(A,[2 3])` sorts the rows of `A` by the second column, and where these are equal, further sorts by the third column.

`[B,index] = sortrows(A)` also returns an index vector `index`.

If `A` is a column vector, then `B = A(index)`.

If `A` is an `m`-by-`n` matrix, then `B = A(index,:)`.

Examples Given the 5-by-5 string matrix,

```
A = ['one  ';'two  ';'three';'four ';'five '];
```

The commands `B = sortrows(A)` and `C = sortrows(A,1)` yield

```
B =         C =
    five        four
    four        five
    one         one
    three       two
    two         three
```

See Also `sort`

Purpose Convert vector into sound

Syntax

```
sound(y,Fs)
sound(y)
sound(y,Fs,bits)
```

Description `sound(y,Fs)`, sends the signal in vector `y` (with sample frequency `Fs`) to the speaker on PC and most UNIX platforms. Values in `y` are assumed to be in the range $-1.0 \leq y \leq 1.0$. Values outside that range are clipped. Stereo sound is played on platforms that support it when `y` is an `n`-by-2 matrix.

`sound(y)` plays the sound at the default sample rate or 8192 Hz.

`sound(y,Fs,bits)` plays the sound using `bits` number of bits/sample, if possible. Most platforms support `bits = 8` or `bits = 16`.

Remarks MATLAB supports all Windows-compatible sound devices.

See Also `auread`, `auwrite`, `soundsc`, `wavread`, `wavwrite`

Purpose

Scale data and play as sound

Syntax

```
soundsc(y,Fs)
soundsc(y)
soundsc(y,Fs,bits)
soundsc(y,...,slim)
```

Description

`soundsc(y,Fs)` sends the signal in vector `y` (with sample frequency `Fs`) to the speaker on PC and most UNIX platforms. The signal `y` is scaled to the range $-1.0 \le y \le 1.0$ before it is played, resulting in a sound that is played as loud as possible without clipping.

`soundsc(y)` plays the sound at the default sample rate or 8192 Hz.

`soundsc(y,Fs,bits)` plays the sound using `bits` number of bits/sample if possible. Most platforms support `bits = 8` or `bits = 16`.

`soundsc(y,...,slim)` where `slim = [slow shigh]` maps the values in `y` between `slow` and `shigh` to the full sound range. The default value is `slim = [min(y) max(y)]`.

Remarks

MATLAB supports all Windows-compatible sound devices.

See Also

`auread`, `auwrite`, `sound`, `wavread`, `wavwrite`

spalloc

Purpose Allocate space for sparse matrix

Syntax

```
S = spalloc(m,n,nzmax)
```

Description

`S = spalloc(m,n,nzmax)` creates an all zero sparse matrix `S` of size `m`-by-`n` with room to hold `nzmax` nonzeros. The matrix can then be generated column by column without requiring repeated storage allocation as the number of nonzeros grows.

`spalloc(m,n,nzmax)` is shorthand for

```
sparse([],[],[],m,n,nzmax)
```

Examples

To generate efficiently a sparse matrix that has an average of at most three nonzero elements per column

```
S = spalloc(n,n,3*n);
for j = 1:n
    S(:,j) = [zeros(n-3,1)' round(rand(3,1))']';
end
```

Purpose

Create sparse matrix

Syntax

```
S = sparse(A)
S = sparse(i,j,s,m,n,nzmax)
S = sparse(i,j,s,m,n)
S = sparse(i,j,s)
S = sparse(m,n)
```

Description

The `sparse` function generates matrices in MATLAB's sparse storage organization.

`S = sparse(A)` converts a full matrix to sparse form by squeezing out any zero elements. If `S` is already sparse, `sparse(S)` returns `S`.

`S = sparse(i,j,s,m,n,nzmax)` uses vectors `i`, `j`, and `s` to generate an `m`-by-`n` sparse matrix such that `S(i(k),j(k)) = s(k)`, with space allocated for `nzmax` nonzeros. Vectors `i`, `j`, and `s` are all the same length. Any elements of `s` that are zero are ignored, along with the corresponding values of `i` and `j`. Any elements of `s` that have duplicate values of `i` and `j` are added together.

Note If any value in `i` or `j` is larger than the maximum integer size, `2^31-1`, then the sparse matrix cannot be constructed.

To simplify this six-argument call, you can pass scalars for the argument `s` and one of the arguments `i` or `j`—in which case they are expanded so that `i`, `j`, and `s` all have the same length.

`S = sparse(i,j,s,m,n)` uses `nzmax = length(s)`.

`S = sparse(i,j,s)` uses `m = max(i)` and `n = max(j)`. The maxima are computed before any zeros in `s` are removed, so one of the rows of `[i j s]` might be `[m n 0]`.

`S = sparse(m,n)` abbreviates `sparse([],[],[],m,n,0)`. This generates the ultimate sparse matrix, an `m`-by-`n` all zero matrix.

Remarks

All of MATLAB's built-in arithmetic, logical, and indexing operations can be applied to sparse matrices, or to mixtures of sparse and full matrices. Operations on sparse matrices return sparse matrices and operations on full matrices return full matrices.

In most cases, operations on mixtures of sparse and full matrices return full matrices. The exceptions include situations where the result of a mixed operation is structurally sparse, for example, `A.*S` is at least as sparse as `S`.

Examples

`S = sparse(1:n,1:n,1)` generates a sparse representation of the `n`-by-`n` identity matrix. The same `S` results from `S = sparse(eye(n,n))`, but this would also temporarily generate a full `n`-by-`n` matrix with most of its elements equal to zero.

`B = sparse(10000,10000,pi)` is probably not very useful, but is legal and works; it sets up a `10000`-by-`10000` matrix with only one nonzero element. Don't try `full(B)`; it requires 800 megabytes of storage.

This dissects and then reassembles a sparse matrix:

```
[i,j,s] = find(S);
[m,n] = size(S);
S = sparse(i,j,s,m,n);
```

So does this, if the last row and column have nonzero entries:

```
[i,j,s] = find(S);
S = sparse(i,j,s);
```

See Also

`diag`, `find`, `full`, `nnz`, `nonzeros`, `nzmax`, `spones`, `sprandn`, `sprandsym`, `spy`

The `sparfun` directory

Purpose Form least squares augmented system

Syntax

```
S = spaugment(A,c)
```

Description

`S = spaugment(A,c)` creates the sparse, square, symmetric indefinite matrix `S = [c*I A; A' 0]`. The matrix `S` is related to the least squares problem

```
min norm(b - A*x)
```

by

```
r = b - A*x
S * [r/c; x] = [b; 0]
```

The optimum value of the residual scaling factor `c`, involves `min(svd(A))` and `norm(r)`, which are usually too expensive to compute.

`S = spaugment(A)` without a specified value of `c`, uses `max(max(abs(A)))/1000`.

Note In previous versions of MATLAB, the augmented matrix was used by sparse linear equation solvers, \ and /, for nonsquare problems. Now, MATLAB performs a least squares solve using the `qr` factorization of `A` instead.

See Also

`spparms`

spconvert

Purpose

Import matrix from sparse matrix external format

Syntax

```
S = spconvert(D)
```

Description

`spconvert` is used to create sparse matrices from a simple sparse format easily produced by non-MATLAB sparse programs. `spconvert` is the second step in the process:

1. Load an ASCII data file containing `[i,j,v]` or `[i,j,re,im]` as rows into a MATLAB variable.
2. Convert that variable into a MATLAB sparse matrix.

`S = spconvert(D)` converts a matrix `D` with rows containing `[i,j,s]` or `[i,j,r,s]` to the corresponding sparse matrix. `D` must have an `nnz` or `nnz+1` row and three or four columns. Three elements per row generate a real matrix and four elements per row generate a complex matrix. A row of the form `[m n 0]` or `[m n 0 0]` anywhere in `D` can be used to specify `size(S)`. If `D` is already sparse, no conversion is done, so `spconvert` can be used after `D` is loaded from either a MAT-file or an ASCII file.

Examples

Suppose the ASCII file `uphill.dat` contains

```
1    1    1.000000000000000
1    2    0.500000000000000
2    2    0.333333333333333
1    3    0.333333333333333
2    3    0.250000000000000
3    3    0.200000000000000
1    4    0.250000000000000
2    4    0.200000000000000
3    4    0.166666666666667
4    4    0.142857142857143
4    4    0.000000000000000
```

Then the statements

```
load uphill.dat
H = spconvert(uphill)
```

```
H =
   (1,1)       1.0000
   (1,2)       0.5000
   (2,2)       0.3333
   (1,3)       0.3333
   (2,3)       0.2500
   (3,3)       0.2000
   (1,4)       0.2500
   (2,4)       0.2000
   (3,4)       0.1667
   (4,4)       0.1429
```

recreate `sparse(triu(hilb(4)))`, possibly with roundoff errors. In this case, the last line of the input file is not necessary because the earlier lines already specify that the matrix is at least 4-by-4.

spdiags

Purpose

Extract and create sparse band and diagonal matrices

Syntax

```
[B,d] = spdiags(A)
B = spdiags(A,d)
A = spdiags(B,d,A)
A = spdiags(B,d,m,n)
```

Description

The spdiags function generalizes the function `diag`. Four different operations, distinguished by the number of input arguments, are possible:

`[B,d] = spdiags(A)` extracts all nonzero diagonals from the `m`-by-`n` matrix `A`. `B` is a `min(m,n)`-by-`p` matrix whose columns are the `p` nonzero diagonals of `A`. `d` is a vector of length `p` whose integer components specify the diagonals in `A`.

`B = spdiags(A,d)` extracts the diagonals specified by `d`.

`A = spdiags(B,d,A)` replaces the diagonals specified by `d` with the columns of `B`. The output is sparse.

`A = spdiags(B,d,m,n)` creates an `m`-by-`n` sparse matrix by taking the columns of `B` and placing them along the diagonals specified by `d`.

Note If a column of `B` is longer than the diagonal it's replacing, `spdiags` takes elements of super-diagonals from the lower part of the column of `B`, and elements of sub-diagonals from the upper part of the column of `B`.

Arguments

The `spdiags` function deals with three matrices, in various combinations, as both input and output.

- `A` An `m`-by-`n` matrix, usually (but not necessarily) sparse, with its nonzero or specified elements located on `p` diagonals.
- `B` A `min(m,n)`-by-`p` matrix, usually (but not necessarily) full, whose columns are the diagonals of `A`.
- `d` A vector of length `p` whose integer components specify the diagonals in `A`.

Roughly, A, B, and d are related by

```
for k = 1:p
    B(:,k) = diag(A,d(k))
end
```

Some elements of B, corresponding to positions outside of A, are not defined by these loops. They are not referenced when B is input and are set to zero when B is output.

Examples

Example 1. This example generates a sparse tridiagonal representation of the classic second difference operator on n points.

```
e = ones(n,1);
A = spdiags([e -2*e e], -1:1, n, n)
```

Turn it into Wilkinson's test matrix (see gallery):

```
A = spdiags(abs(-(n-1)/2:(n-1)/2)',0,A)
```

Finally, recover the three diagonals:

```
B = spdiags(A)
```

Example 2. The second example is not square.

```
A = [11    0   13    0
      0   22    0   24
      0    0   33    0
     41    0    0   44
      0   52    0    0
      0    0   63    0
      0    0    0   74]
```

Here m = 7, n = 4, and p = 3.

The statement [B,d] = spdiags(A) produces d = [-3 0 2]' and

```
B = [41   11    0
     52   22    0
     63   33   13
     74   44   24]
```

Conversely, with the above `B` and `d`, the expression `spdiags(B,d,7,4)` reproduces the original `A`.

Example 3. This example shows how `spdiags` creates the diagonals when the columns of `B` are longer than the diagonals they are replacing.

```
B = repmat((1:6)',[1 7])

B =

    1   1   1   1   1   1   1
    2   2   2   2   2   2   2
    3   3   3   3   3   3   3
    4   4   4   4   4   4   4
    5   5   5   5   5   5   5
    6   6   6   6   6   6   6

d = [-4 -2 -1 0 3 4 5];
A = spdiags(B,d,6,6);
full(A)

ans =

  1   0   0   4   5   6
  1   2   0   0   5   6
  1   2   3   0   0   6
  0   2   3   4   0   0
  1   0   3   4   5   0
  0   2   0   4   5   6
```

See Also

`diag`

Purpose Sparse identity matrix

Syntax

```
S = speye(m,n)
S = speye(n)
```

Description `S = speye(m,n)` forms an `m`-by-`n` sparse matrix with `1`s on the main diagonal.

`S = speye(n)` abbreviates `speye(n,n)`.

Examples `I = speye(1000)` forms the sparse representation of the 1000-by-1000 identity matrix, which requires only about 16 kilobytes of storage. This is the same final result as `I = sparse(eye(1000,1000))`, but the latter requires eight megabytes for temporary storage for the full representation.

See Also `spalloc`, `spones`, `spdiags`, `sprand`, `sprandn`

spfun

Purpose

Apply function to nonzero sparse matrix elements

Syntax

```
f = spfun(fun,S)
```

Description

The `spfun` function selectively applies a function to only the *nonzero* elements of a sparse matrix `S`, preserving the sparsity pattern of the original matrix (except for underflow or if `fun` returns zero for some nonzero elements of `S`).

`f = spfun(fun,S)` evaluates `fun(S)` on the nonzero elements of `S`. You can specify `fun` as a function handle or as an inline object.

Remarks

Functions that operate element-by-element, like those in the `elfun` directory, are the most appropriate functions to use with `spfun`.

Examples

Given the 4-by-4 sparse diagonal matrix

```
S = spdiags([1:4]',0,4,4)

S =
   (1,1)        1
   (2,2)        2
   (3,3)        3
   (4,4)        4
```

Because `fun` returns nonzero values for all nonzero element of `S`, `f = spfun(@exp,S)` has the same sparsity pattern as `S`.

```
f =
   (1,1)        2.7183
   (2,2)        7.3891
   (3,3)       20.0855
   (4,4)       54.5982
```

whereas `exp(S)` has `1`s where `S` has `0`s.

```
full(exp(S))

ans =
    2.7183    1.0000    1.0000    1.0000
    1.0000    7.3891    1.0000    1.0000
```

```
1.0000    1.0000   20.0855    1.0000
1.0000    1.0000    1.0000   54.5982
```

See Also

function handle (@), `inline`

sph2cart

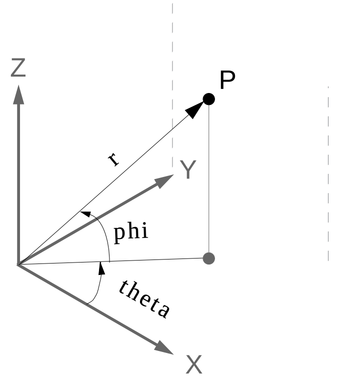
Purpose Transform spherical coordinates to Cartesian

Syntax

```
[x,y,z] = sph2cart(THETA,PHI,R)
```

Description `[x,y,z] = sph2cart(THETA,PHI,R)` transforms the corresponding elements of spherical coordinate arrays to Cartesian, or *xyz*, coordinates. `THETA`, `PHI`, and `R` must all be the same size. `THETA` and `PHI` are angular displacements in radians from the positive *x*-axis and from the *x*-*y* plane, respectively.

Algorithm The mapping from spherical coordinates to three-dimensional Cartesian coordinates is




```
x = r .* cos(phi) .* cos(theta)
y = r .* cos(phi) .* sin(theta)
      z = r .* sin(phi)
```

See Also `cart2pol`, `cart2sph`, `pol2cart`

Purpose Generate sphere

Syntax

```
sphere
sphere(n)
[X,Y,Z] = sphere(...)
```

Description The `sphere` function generates the *x*-, *y*-, and *z*-coordinates of a unit sphere for use with `surf` and `mesh`.

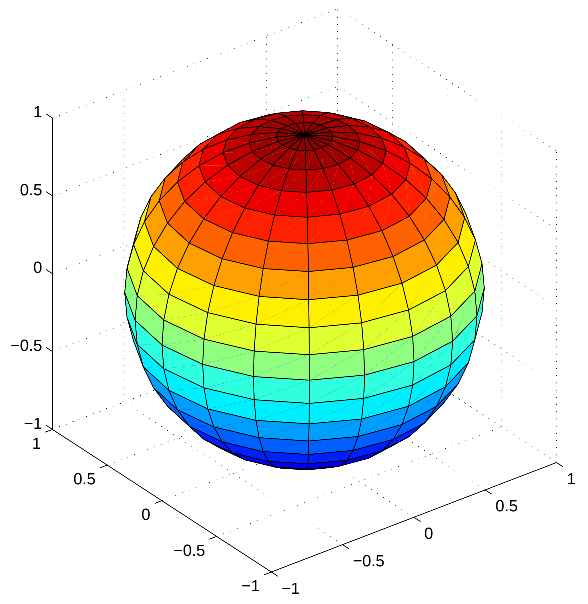
`sphere` generates a sphere consisting of 20-by-20 faces.

`sphere(n)` draws a `surf` plot of an `n`-by-`n` sphere in the current figure.

`[X,Y,Z] = sphere(n)` returns the coordinates of a sphere in three matrices that are `(n+1)`–by–`(n+1)` in size. You draw the sphere with `surf(X,Y,Z)` or `mesh(X,Y,Z)`.

Examples Generate and plot a sphere.

```
sphere
axis equal
```



See Also `cylinder`, `axis`

Purpose

Spin colormap

Syntax

```
spinmap
spinmap(t)
spinmap(t,inc)
spinmap('inf')
```

Description

The `spinmap` function shifts the colormap RGB values by some incremental value. For example, if the increment equals 1, color 1 becomes color 2, color 2 becomes color 3, etc.

`spinmap` cyclically rotates the colormap for approximately five seconds using an incremental value of 2.

`spinmap(t)` rotates the colormap for approximately `10*t` seconds. The amount of time specified by `t` depends on your hardware configuration (e.g., if you are running MATLAB over a network).

`spinmap(t,inc)` rotates the colormap for approximately `10*t` seconds and specifies an increment `inc` by which the colormap shifts. When `inc` is 1, the rotation appears smoother than the default (i.e., 2). Increments greater than 2 are less smooth than the default. A negative increment (e.g., –2) rotates the colormap in a negative direction.

`spinmap('inf')` rotates the colormap for an infinite amount of time. To break the loop, press **Ctrl-C**.

See Also

`colormap`

spline

Purpose

Cubic spline data interpolation

Syntax

```
yy = spline(x,y,xx)
pp = spline(x,y)
```

Description

`yy = spline(x,y,xx)` uses cubic spline interpolation to find `yy`, the values of the underlying function `y` at the points in the vector `xx`. The vector `x` specifies the points at which the data `y` is given. If `y` is a matrix, then the data is taken to be vector-valued and interpolation is performed for each column of `y` and `yy` is `length(xx)`-by-`size(y,2)`.

`pp = spline(x,y)` returns the piecewise polynomial form of the cubic spline interpolant for later use with `ppval` and the spline utility `unmkpp`.

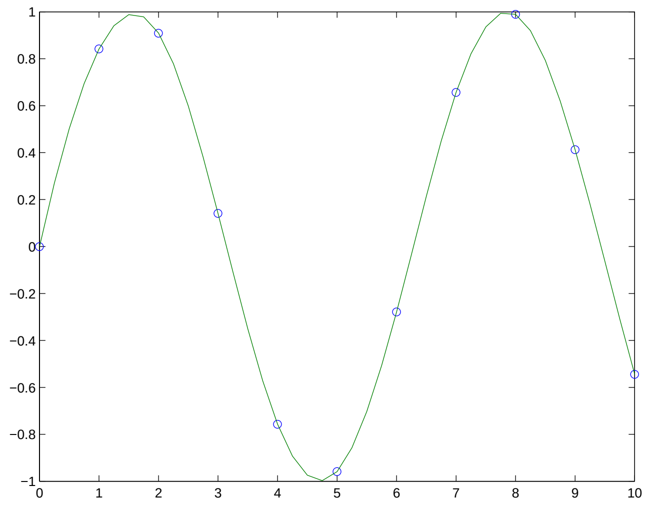
Ordinarily, the not-a-knot end conditions are used. However, if `y` contains two more values than `x` has entries, then the first and last value in `y` are used as the endslopes for the cubic spline. Namely:

```
f(x) = y(:,2:end-1), df(min(x)) = y(:,1), df(max(x)) = y(:,end)
```

Examples

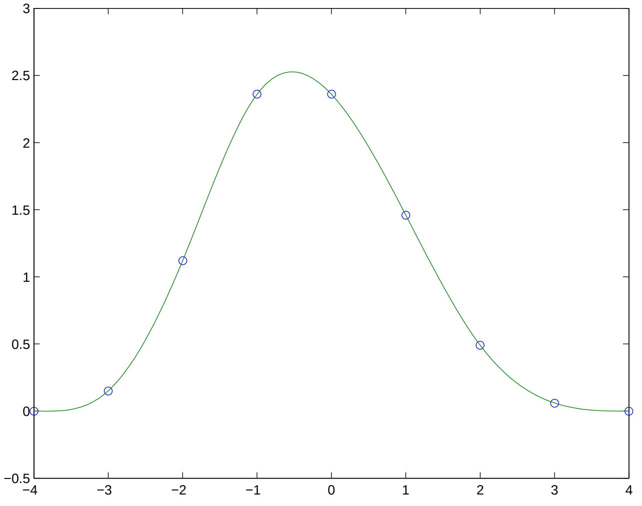
Example 1. This generates a sine curve, then samples the spline over a finer mesh.

```
x = 0:10;
y = sin(x);
xx = 0:.25:10;
yy = spline(x,y,xx);
plot(x,y,'o',xx,yy)
```



Example 2. This illustrates the use of clamped or complete spline interpolation where end slopes are prescribed. Zero slopes at the ends of an interpolant to the values of a certain distribution are enforced.

```
x = -4:4;
y = [0 .15 1.12 2.36 2.36 1.46 .49 .06 0];
cs = spline(x,[0 y 0]);
xx = linspace(-4,4,101);
plot(x,y,'o',xx,ppval(cs,xx),'-');
```



Example 3. The two vectors

```
t = 1900:10:1990;
p = [ 75.995  91.972  105.711  123.203  131.669 ...
     150.697 179.323  203.212  226.505  249.633 ];
```

represent the census years from 1900 to 1990 and the corresponding United States population in millions of people. The expression

```
spline(t,p,2000)
```

uses the cubic spline to extrapolate and predict the population in the year 2000. The result is

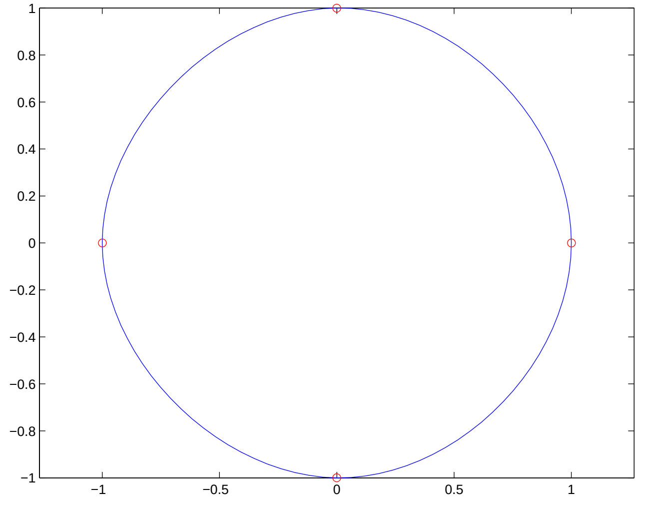
```
ans =
    270.6060
```

Example 4. The statements

```
x = pi*[0:.5:2];
y = [0  1  0 -1  0  1  0;
     1  0  1  0 -1  0  1];
pp = spline(x,y);
```

```
yy = ppval(pp, linspace(0,2*pi,101));
plot(yy(1,:),yy(2,:),'-b',y(1,2:5),y(2,2:5),'or'), axis equal
```

generate the plot of a circle, with the five data points `y(:,2),...,y(:,6)` marked with o's. Note that this `y` contains two more values (i.e., two more columns) than does `x`, hence `y(:,1)` and `y(:,end)` are used as endslopes.



Algorithm

A tridiagonal linear system (with, possibly, several right sides) is being solved for the information needed to describe the coefficients of the various cubic polynomials which make up the interpolating spline. `spline` uses the functions `ppval`, `mkpp`, and `unmkpp`. These routines form a small suite of functions for working with piecewise polynomials. For access to more advanced features, see the M-file help for these functions and the Spline Toolbox.

See Also

`interp1`, `ppval`, `mkpp`, `unmkpp`

References

[1] de Boor, C., *A Practical Guide to Splines*, Springer-Verlag, 1978.

spones

Purpose

Replace nonzero sparse matrix elements with ones

Syntax

```
R = spones(S)
```

Description

`R = spones(S)` generates a matrix `R` with the same sparsity structure as `S`, but with `1`'s in the nonzero positions.

Examples

`c = sum(spones(S))` is the number of nonzeros in each column.

`r = sum(spones(S'))'` is the number of nonzeros in each row.

`sum(c)` and `sum(r)` are equal, and are equal to `nnz(S)`.

See Also

`nnz`, `spalloc`, `spfun`

Purpose Set parameters for sparse matrix routines

Syntax

```
spparms('key',value)
spparms
values = spparms
[keys,values] = spparms
spparms(values)
value = spparms('key')
spparms('default')
spparms('tight')
```

Description `spparms('key',value)` sets one or more of the *tunable* parameters used in the sparse linear equation operators, `\` and `/`, and the minimum degree orderings, `colmmd` and `symmmd`. In ordinary use, you should never need to deal with this function.

The meanings of the `key` parameters are

`'spumoni'`	Sparse Monitor flag. `0` produces no diagnostic output, the default. `1` produces information about choice of algorithm based on matrix structure, and about storage allocation. `2` also produces very detailed information about the minimum degree algorithms.
`'thr_rel'`, `'thr_abs'`	Minimum degree threshold is `thr_rel*mindegree+thr_abs`.
`'exact_d'`	Nonzero to use exact degrees in minimum degree. Zero to use approximate degrees.
`'supernd'`	If positive, minimum degree amalgamates the supernodes every `supernd` stages.
`'rreduce'`	If positive, minimum degree does row reduction every `rreduce` stages.
`'wh_frac'`	Rows with `density > wh_frac` are ignored in `colmmd`.

`'autommd'`	Nonzero to use minimum degree orderings with `\` and `/`.
`'aug_rel'`, `'aug_abs'`	Residual scaling parameter for augmented equations is `aug_rel*max(max(abs(A))) + aug_abs`. For example, `aug_rel = 0`, `aug_abs = 1` puts an unscaled identity matrix in the (1,1) block of the augmented matrix.

`spparms`, by itself, prints a description of the current settings.

`values = spparms` returns a vector whose components give the current settings.

`[keys,values] = spparms` returns that vector, and also returns a character matrix whose rows are the keywords for the parameters.

`spparms(values)`, with no output argument, sets all the parameters to the values specified by the argument vector.

`value = spparms('key')` returns the current setting of one parameter.

`spparms('default')` sets all the parameters to their default settings.

`spparms('tight')` sets the minimum degree ordering parameters to their *tight* settings, which can lead to orderings with less fill-in, but which make the ordering functions themselves use more execution time.

The key parameters for `default` and `tight` settings are

	Keyword	**Default**	**Tight**
`values(1)`	`'spumoni'`	0.0	
`values(2)`	`'thr_rel'`	1.1	1.0
`values(3)`	`'thr_abs'`	1.0	0.0
`values(4)`	`'exact_d'`	0.0	1.0
`values(5)`	`'supernd'`	3.0	1.0
`values(6)`	`'rreduce'`	3.0	1.0

	Keyword	Default	Tight
`values(7)`	`'wh_frac'`	0.5	0.5
`values(8)`	`'autommd'`	1.0	
`values(9)`	`'aug_rel'`	0.001	
`values(10)`	`'aug_abs'`	0.0	

See Also

`\`, `colamd`, `colmmd`, `symamd`, `symmmd`

References

[1] Gilbert, John R., Cleve Moler and Robert Schreiber, "Sparse Matrices in MATLAB: Design and Implementation," *SIAM Journal on Matrix Analysis and Applications*, Vol. 13, 1992, pp. 333-356.

Purpose Sparse uniformly distributed random matrix

Syntax

```
R = sprand(S)
R = sprand(m,n,density)
R = sprand(m,n,density,rc)
```

Description

`R = sprand(S)` has the same sparsity structure as `S`, but uniformly distributed random entries.

`R = sprand(m,n,density)` is a random, `m`-by-`n`, sparse matrix with approximately `density*m*n` uniformly distributed nonzero entries (`0 <= density <= 1`).

`R = sprand(m,n,density,rc)` also has reciprocal condition number approximately equal to `rc`. `R` is constructed from a sum of matrices of rank one.

If `rc` is a vector of length `lr`, where `lr <= min(m,n)`, then `R` has `rc` as its first `lr` singular values, all others are zero. In this case, `R` is generated by random plane rotations applied to a diagonal matrix with the given singular values. It has a great deal of topological and algebraic structure.

See Also `sprandn`, `sprandsym`

Purpose Sparse normally distributed random matrix

Syntax

```
R = sprandn(S)
R = sprandn(m,n,density)
R = sprandn(m,n,density,rc)
```

Description

`R = sprandn(S)` has the same sparsity structure as `S`, but normally distributed random entries with mean `0` and variance `1`.

`R = sprandn(m,n,density)` is a random, `m`-by-`n`, sparse matrix with approximately `density*m*n` normally distributed nonzero entries (`0 <= density <= 1`).

`R = sprandn(m,n,density,rc)` also has reciprocal condition number approximately equal to `rc`. `R` is constructed from a sum of matrices of rank one.

If `rc` is a vector of length `lr`, where `lr <= min(m,n)`, then `R` has `rc` as its first `lr` singular values, all others are zero. In this case, `R` is generated by random plane rotations applied to a diagonal matrix with the given singular values. It has a great deal of topological and algebraic structure.

See Also `sprand`, `sprandsym`

sprandsym

Purpose Sparse symmetric random matrix

Syntax

```
R = sprandsym(S)
R = sprandsym(n,density)
R = sprandsym(n,density,rc)
R = sprandsym(n,density,rc,kind)
```

Description `R = sprandsym(S)` returns a symmetric random matrix whose lower triangle and diagonal have the same structure as `S`. Its elements are normally distributed, with mean `0` and variance `1`.

`R = sprandsym(n,density)` returns a symmetric random, `n`-by-`n`, sparse matrix with approximately `density*n*n` nonzeros; each entry is the sum of one or more normally distributed random samples, and `(0 <= density <= 1)`.

`R = sprandsym(n,density,rc)` returns a matrix with a reciprocal condition number equal to `rc`. The distribution of entries is nonuniform; it is roughly symmetric about 0; all are in $[-1, 1]$.

If `rc` is a vector of length `n`, then `R` has eigenvalues `rc`. Thus, if `rc` is a positive (nonnegative) vector then `R` is a positive definite matrix. In either case, `R` is generated by random Jacobi rotations applied to a diagonal matrix with the given eigenvalues or condition number. It has a great deal of topological and algebraic structure.

`R = sprandsym(n,density,rc,kind)` returns a positive definite matrix. Argument `kind` can be:

- `1` to generate `R` by random Jacobi rotation of a positive definite diagonal matrix. `R` has the desired condition number exactly.
- `2` to generate an `R` that is a shifted sum of outer products. `R` has the desired condition number only approximately, but has less structure.
- `3` to generate an `R` that has the same structure as the matrix `S` and approximate condition number `1/rc`. `density` is ignored.

See Also `sprand`, `sprandn`

Purpose Structural rank

Syntax

```
r = sprank(A)
```

Description `r = sprank(A)` is the structural rank of the sparse matrix `A`. Also known as maximum traversal, maximum assignment, and size of a maximum matching in the bipartite graph of `A`.

Always `sprank(A) >= rank(full(A))`, and in exact arithmetic `sprank(A) == rank(full(sprandn(A)))` with probability one.

Examples

```
A = [1    0    2    0
     2    0    4    0 ];

A = sparse(A);

sprank(A)

ans =
     2

rank(full(A))

ans =
     1
```

See Also `dmperm`

sprintf

Purpose

Write formatted data to a string

Syntax

```
[s,errmsg] = sprintf(format,A,...)
```

Description

`[s,errmsg] = sprintf(format,A,...)` formats the data in matrix `A` (and in any additional matrix arguments) under control of the specified `format` string, and returns it in the MATLAB string variable `s`. The `sprintf` function returns an error message string `errmsg` if an error occurred. `errmsg` is an empty matrix if no error occurred.

`sprintf` is the same as `fprintf` except that it returns the data in a MATLAB string variable rather than writing it to a file.

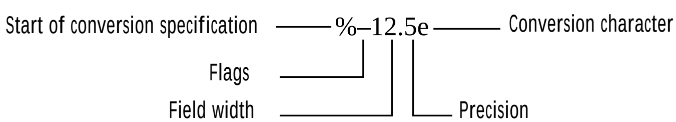
Format String

The `format` argument is a string containing C language conversion specifications. A conversion specification controls the notation, alignment, significant digits, field width, and other aspects of output format. The format string can contain escape characters to represent non-printing characters such as newline characters and tabs.

Conversion specifications begin with the `%` character and contain these optional and required elements:

- Flags (optional)
- Width and precision fields (optional)
- A subtype specifier (optional)
- Conversion character (required)

You specify these elements in the following order:



Flags

You can control the alignment of the output using any of these optional flags.

Character	Description	Example
A minus sign (–)	Left-justifies the converted argument in its field.	`%–5.2d`
A plus sign (+)	Always prints a sign character (+ or –).	`%+5.2d`
Zero (`0`)	Pad with zeros rather than spaces.	`%05.2d`

Field Width and Precision Specifications

You can control the width and precision of the output by including these options in the format string.

Character	Description	Example
Field width	A digit string specifying the minimum number of digits to be printed.	`%6f`
Precision	A digit string including a period (.) specifying the number of digits to be printed to the right of the decimal point.	`%6.2f`

Conversion Characters

Conversion characters specify the notation of the output.

Specifier	Description
`%c`	Single character
`%d`	Decimal notation (signed)
`%e`	Exponential notation (using a lowercase `e` as in `3.1415e+00`)
`%E`	Exponential notation (using an uppercase `E` as in `3.1415E+00`)

Specifier	Description
%f	Fixed-point notation
%g	The more compact of %e or %f, as defined in [2]. Insignificant zeros do not print.
%G	Same as %g, but using an uppercase E
%o	Octal notation (unsigned)
%s	String of characters
%u	Decimal notation (unsigned)
%x	Hexadecimal notation (using lowercase letters a–f)
%X	Hexadecimal notation (using uppercase letters A–F)

The following tables describe the nonalphanumeric characters found in format specification strings.

Escape Characters

This table lists the escape character sequences you use to specify non-printing characters in a format specification.

Character	Description
\b	Backspace
\f	Form feed
\n	New line
\r	Carriage return
\t	Horizontal tab
\\	Backslash

Character	Description
\'' or '' (two single quotes)	Single quotation mark
%%	Percent character

Remarks

The `sprintf` function behaves like its ANSI C language namesake with these exceptions and extensions.

- If you use `sprintf` to convert a MATLAB double into an integer, and the double contains a value that cannot be represented as an integer (for example, it contains a fraction), MATLAB ignores the specified conversion and outputs the value in exponential format. To successfully perform this conversion, use the `fix`, `floor`, `ceil`, or `round` functions to change the value in the double into a value that can be represented as an integer before passing it to `sprintf`.
- The following, non-standard subtype specifiers are supported for the conversion characters `%o`, `%u`, `%x`, and `%X`.

b	The underlying C data type is a double rather than an unsigned integer. For example, to print a double-precision value in hexadecimal, use a format like '`%bx`'.
t	The underlying C data type is a float rather than an unsigned integer.

  For example, to print a double value in hexadecimal use the format `'%bx'`

- The `sprintf` function is vectorized for nonscalar arguments. The function recycles the format string through the elements of `A` (columnwise) until all the elements are used up. The function then continues in a similar manner through any additional matrix arguments.
- If `%s` is used to print part of a nonscalar double argument, the following behavior occurs:

  a. Successive values are printed as long as they are integers and in the range of a valid character. The first invalid character terminates the printing for this `%s` specifier and is used for a later specifier. For example, `pi` terminates the string below and is printed using `%f` format.

```
Str = [65 66 67 pi];
sprintf('%s %f', Str)
ans =
ABC 3.141593
```

b. If the first value to print is not a valid character, then just that value is printed for this `%s` specifier using an `e` conversion as a warning to the user. For example, `pi` is formatted by `%s` below in exponential notation, and `65`, though representing a valid character, is formatted as fixed-point (`%f`).

```
Str = [pi 65 66 67];
sprintf('%s %f %s', Str)
ans =
3.141593e+000 65.000000 BC
```

c. One exception is zero which is a valid character. If zero is found first, `%s` prints nothing and the value is skipped. If zero is found after at least one valid character, it terminates the printing for this `%s` specifier and is used for a later specifier.

- `sprintf` prints negative zero and exponents differently on some platforms, as shown in the following tables.

Negative Zero Printed with %e, %E, %f, %g, or %G

	Display of Negative Zero		
Platform	**%e or %E**	**%f**	**%g or %G**
PC	0.000000e+000	0.000000	0
SGI	0.000000e+00	0.000000	0
HP700	-0.000000e+00	-0.000000	0
Others	-0.000000e+00	-0.000000	-0

Exponents Printed with %e, %E, %g, or %G

Platform	Minimum Digits in Exponent	Example
PC	3	`1.23e+004`
UNIX	2	`1.23e+04`

You can resolve this difference in exponents by post-processing the results of `sprintf`. For example, to make the PC output look like that of UNIX, use

```
a = sprintf('%e', 12345.678);
if ispc, a = strrep(a, 'e+0', 'e+');   end
```

Examples

Command	Result
`sprintf('%0.5g',(1+sqrt(5))/2)`	`1.618`
`sprintf('%0.5g',1/eps)`	`4.5036e+15`
`sprintf('%15.5f',1/eps)`	`4503599627370496.00000`
`sprintf('%d',round(pi))`	`3`
`sprintf('%s','hello')`	`hello`
`sprintf('The array is %dx%d.',2,3)`	`The array is 2x3`
`sprintf('\n')`	Line termination character on all platforms

See Also

`int2str`, `num2str`, `sscanf`

References

[1] Kernighan, B.W. and D.M. Ritchie, *The C Programming Language*, Second Edition, Prentice-Hall, Inc., 1988.

[2] ANSI specification X3.159-1989: "Programming Language C," ANSI, 1430 Broadway, New York, NY 10018.

spy

Purpose Visualize sparsity pattern

Syntax

```
spy(S)
spy(S,markersize)
spy(S,'LineSpec')
spy(S,'LineSpec',markersize)
```

Description `spy(S)` plots the sparsity pattern of any matrix `S`.

`spy(S,markersize)`, where `markersize` is an integer, plots the sparsity pattern using markers of the specified point size.

`spy(S,'LineSpec')`, where `LineSpec` is a string, uses the specified plot marker type and color.

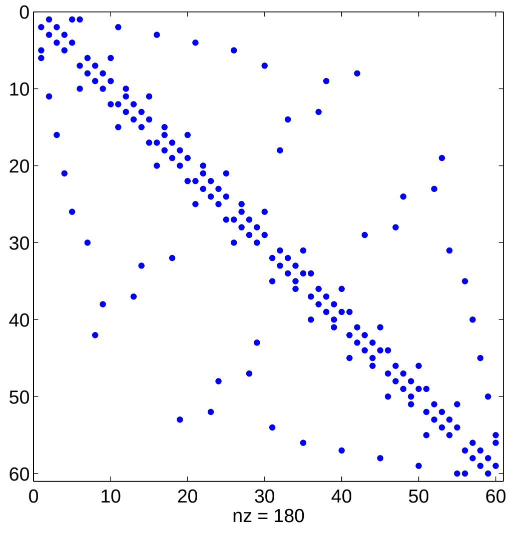
`spy(S,'LineSpec',markersize)` uses the specified type, color, and size for the plot markers.

`S` is usually a sparse matrix, but full matrices are acceptable, in which case the locations of the nonzero elements are plotted.

Note spy replaces `format +`, which takes much more space to display essentially the same information.

Examples This example plots the 60-by-60 sparse adjacency matrix of the connectivity graph of the Buckminster Fuller geodesic dome. This matrix also represents the soccer ball and the carbon-60 molecule.

```
B = bucky;
spy(B)
```



See Also

`find`, `gplot`, `LineSpec`, `symamd`, `symmmd`, `symrcm`

sqrt

Purpose Square root

Syntax

```
B = sqrt(X)
```

Description `B = sqrt(X)` returns the square root of each element of the array `X`. For the elements of `X` that are negative or complex, `sqrt(X)` produces complex results.

Remarks See `sqrtm` for the matrix square root.

Examples

```
sqrt((-2:2)')
ans =
      0 + 1.4142i
      0 + 1.0000i
      0
 1.0000
 1.4142
```

See Also `sqrtm`

Purpose

Matrix square root

Syntax

```
X = sqrtm(A)
[X,resnorm] = sqrtm(A)
[X,alpha,condest] = sqrtm(A)
```

Description

`X = sqrtm(A)` is the principal square root of the matrix `A`, i.e. `X*X = A`.

`X` is the unique square root for which every eigenvalue has nonnegative real part. If `A` has any eigenvalues with negative real parts then a complex result is produced. If `A` is singular then `A` may not have a square root. A warning is printed if exact singularity is detected.

`[X, resnorm] = sqrtm(A)` does not print any warning, and returns the residual, `norm(A-X^2,'fro')/norm(A,'fro')`.

`[X, alpha, condest] = sqrtm(A)` returns a stability factor `alpha` and an estimate `condest` of the matrix square root condition number of `X`. The residual `norm(A-X^2,'fro')/norm(A,'fro')` is bounded approximately by `n*alpha*eps` and the Frobenius norm relative error in `X` is bounded approximately by `n*alpha*condest*eps`, where `n = max(size(A))`.

Remarks

If `X` is real, symmetric and positive definite, or complex, Hermitian and positive definite, then so is the computed matrix square root.

Some matrices, like `X = [0 1; 0 0]`, do not have any square roots, real or complex, and `sqrtm` cannot be expected to produce one.

Examples

Example 1. A matrix representation of the fourth difference operator is

```
X =
     5    -4     1     0     0
    -4     6    -4     1     0
     1    -4     6    -4     1
     0     1    -4     6    -4
     0     0     1    -4     5
```

This matrix is symmetric and positive definite. Its unique positive definite square root, `Y = sqrtm(X)`, is a representation of the second difference operator.

```
Y =
     2    -1    -0    -0    -0
    -1     2    -1     0    -0
     0    -1     2    -1     0
    -0     0    -1     2    -1
    -0    -0    -0    -1     2
```

Example 2. The matrix

```
X =
     7    10
    15    22
```

has four square roots. Two of them are

```
Y1 =
    1.5667    1.7408
    2.6112    4.1779
```

and

```
Y2 =
     1     2
     3     4
```

The other two are -Y1 and -Y2. All four can be obtained from the eigenvalues and vectors of X.

```
[V,D] = eig(X);
D =
    0.1386          0
         0    28.8614
```

The four square roots of the diagonal matrix D result from the four choices of sign in

```
S =
     0.3723          0
          0     5.3723
```

All four Ys are of the form

```
Y = V*S/V
```

The `sqrtm` function chooses the two plus signs and produces `Y1`, even though `Y2` is more natural because its entries are integers.

See Also

`expm`, `funm`, `logm`

squeeze

Purpose Remove singleton dimensions

Syntax

```
B = squeeze(A)
```

Description

`B = squeeze(A)` returns an array `B` with the same elements as `A`, but with all singleton dimensions removed. A singleton dimension is any dimension for which `size(A,dim) = 1`.

Examples

Consider the 2-by-1-by-3 array `Y = rand(2,1,3)`. This array has a singleton column dimension — that is, there's only one column per page.

```
Y =

Y(:,:,1) =      Y(:,:,2) =
      0.5194          0.0346
      0.8310          0.0535

Y(:,:,3) =
      0.5297
      0.6711
```

The command `Z = squeeze(Y)` yields a 2-by-3 matrix:

```
Z =
    0.5194    0.0346    0.5297
    0.8310    0.0535    0.6711
```

See Also

`reshape`, `shiftdim`

Purpose Read string under format control

Syntax

```
A = sscanf(s,format)
A = sscanf(s,format,size)
[A,count,errmsg,nextindex] = sscanf(...)
```

Description

`A = sscanf(s,format)` reads data from the MATLAB string variable `s`, converts it according to the specified `format` string, and returns it in matrix `A`. `format` is a string specifying the format of the data to be read. See "Remarks" for details. `sscanf` is the same as `fscanf` except that it reads the data from a MATLAB string variable rather than reading it from a file.

`A = sscanf(s,format,size)` reads the amount of data specified by `size` and converts it according to the specified `format` string. `size` is an argument that determines how much data is read. Valid options are

`n`	Read `n` elements into a column vector.
`inf`	Read to the end of the file, resulting in a column vector containing the same number of elements as are in the file.
`[m,n]`	Read enough elements to fill an `m`-by-`n` matrix, filling the matrix in column order. `n` can be `Inf`, but not `m`.

If the matrix `A` results from using character conversions only and `size` is not of the form `[M,N]`, a row vector is returned.

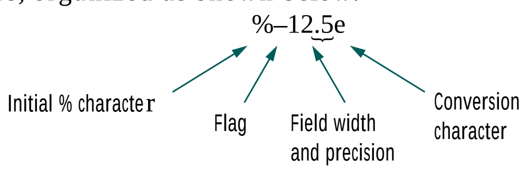
`sscanf` differs from its C language namesakes `scanf()` and `fscanf()` in an important respect — it is *vectorized* in order to return a matrix argument. The `format` string is cycled through the file until an end-of-file is reached or the amount of data specified by `size` is read in.

`[A,count,errmsg,nextindex] = sscanf(...)` reads data from the MATLAB string variable `s`, converts it according to the specified `format` string, and returns it in matrix `A`. `count` is an optional output argument that returns the number of elements successfully read. `errmsg` is an optional output argument that returns an error message string if an error occurred or an empty matrix if an error did not occur. `nextindex` is an optional output argument specifying one more than the number of characters scanned in `s`.

Remarks

When MATLAB reads a specified file, it attempts to match the data in the file to the format string. If a match occurs, the data is written into the matrix in column order. If a partial match occurs, only the matching data is written to the matrix, and the read operation stops.

The `format` string consists of ordinary characters and/or conversion specifications. Conversion specifications indicate the type of data to be matched and involve the character `%`, optional width fields, and conversion characters, organized as shown below:



Add one or more of these characters between the `%` and the conversion character.

An asterisk (*)	Skip over the matched value if the value is matched but not stored in the output matrix.
A digit string	Maximum field width.
A letter	The size of the receiving object; for example, `h` for short as in `%hd` for a short integer, or `l` for long as in `%ld` for a long integer or `%lg` for a double floating-point number.

Valid conversion characters are as shown.

`%c`	Sequence of characters; number specified by field width
`%d`	Decimal numbers
`%e`, `%f`, `%g`	Floating-point numbers
`%i`	Signed integer
`%o`	Signed octal integer
`%s`	A series of non-whitespace characters

`%u`	Signed decimal integer
`%x`	Signed hexadecimal integer
`[...]`	Sequence of characters (scanlist)

If `%s` is used, an element read may use several MATLAB matrix elements, each holding one character. Use `%c` to read space characters, or `%s` to skip all white space.

Mixing character and numeric conversion specifications cause the resulting matrix to be numeric and any characters read to appear as their ASCII values, one character per MATLAB matrix element.

For more information about format strings, refer to the `scanf()` and `fscanf()` routines in a C language reference manual.

Examples

The statements

```
s = '2.7183  3.1416';
A = sscanf(s,'%f')
```

create a two-element vector containing poor approximations to `e` and `pi`.

See Also

`eval`, `sprintf`, `textread`

stairs

Purpose

Stairstep plot

Syntax

```
stairs(Y)
stairs(X,Y)
stairs(...,LineSpec)
[xb,yb] = stairs(Y)
[xb,yb] = stairs(X,Y)
```

Description

Stairstep plots are useful for drawing time-history plots of digitally sampled data systems.

`stairs(Y)` draws a stairstep plot of the elements of `Y`. When `Y` is a vector, the *x*-axis scale ranges from `1` to `size(Y)`. When `Y` is a matrix, the *x*-axis scale ranges from `1` to the number of rows in `Y`.

`stairs(X,Y)` plots `X` versus the columns of `Y`. `X` and `Y` are vectors of the same size or matrices of the same size. Additionally, `X` can be a row or a column vector, and `Y` a matrix with `length(X)` rows.

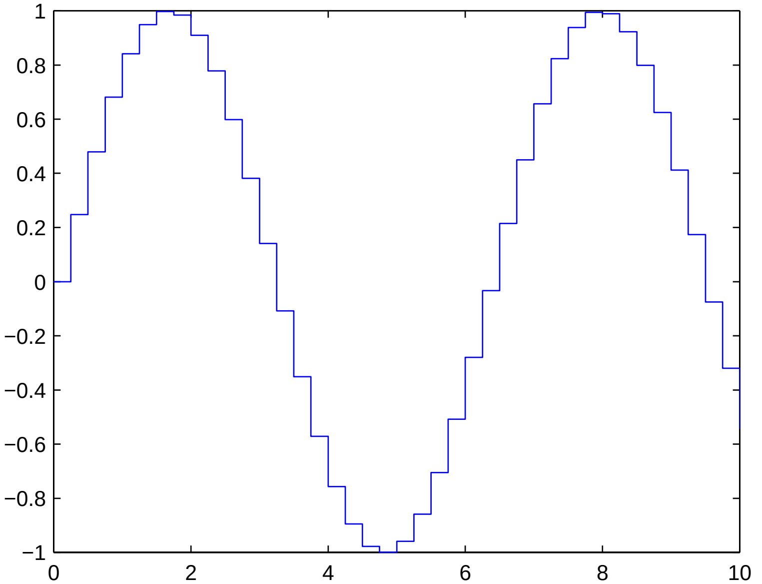
`stairs(...,LineSpec)` specifies a line style, marker symbol, and color for the plot (see `LineSpec` for more information).

`[xb,yb] = stairs(Y)` and `[xb,yb] = stairs(x,Y)` do not draw graphs, but return vectors `xb` and `yb` such that `plot(xb,yb)` plots the stairstep graph.

Examples

Create a stairstep plot of a sine wave.

```
x = 0:.25:10;
stairs(x,sin(x))
```



See Also `bar`, `hist`

startup

Purpose

MATLAB startup M-file for user-defined options

Description

`startup` automatically executes the master M-file `matlabrc.m` and, if it exists, `startup.m`, when MATLAB starts. On multiuser or networked systems, `matlabrc.m` is reserved for use by the system manager. The file `matlabrc.m` invokes the file `startup.m` if it exists on MATLAB's search path.

You can create a `startup.m` file in your own MATLAB directory. The file can include physical constants, handle graphics defaults, engineering conversion factors, or anything else you want predefined in your workspace.

There are other way to predefine aspects of MATLAB. See "Startup Options" and "Setting Preferences" in *Using MATLAB*.

Algorithm

Only `matlabrc.m` is actually invoked by MATLAB at startup. However, `matlabrc.m` contains the statements

```
if exist('startup')==2
    startup
end
```

that invoke `startup.m`. You can extend this process to create additional startup M-files, if required.

See Also

`matlabrc`, `quit`

Purpose Standard deviation

Syntax

```
s = std(X)
s = std(X,flag)
s = std(X,flag,dim)
```

Definition There are two common textbook definitions for the standard deviation `s` of a data vector `X`.

$$(1)\quad s = \left(\frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})^2\right)^{\frac{1}{2}}$$

$$(2)\quad s = \left(\frac{1}{n}\sum_{i=1}^{n}(x_i - \bar{x})^2\right)^{\frac{1}{2}}$$

where

$$\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i$$

and n is the number of elements in the sample. The two forms of the equation differ only in $n-1$ versus n in the divisor.

Description `s = std(X)`, where `X` is a vector, returns the standard deviation using (1) above. If `X` is a random sample of data from a normal distribution, s^2 is the best *unbiased* estimate of its variance.

If `X` is a matrix, `std(X)` returns a row vector containing the standard deviation of the elements of each column of `X`. If `X` is a multidimensional array, `std(X)` is the standard deviation of th elements along the first nonsingleton dimension of `X`.

`s = std(X,flag)` for `flag = 0`, is the same as `std(X)`. For `flag = 1`, `std(X,1)` returns the standard deviation using (2) above, producing the second moment of the sample about its mean.

s = std(X,flag,dim) computes the standard deviations along the dimension of X specified by scalar dim.

Examples

For matrix X

```
X =
     1     5     9
     7    15    22

s = std(X,0,1)
s =
   4.2426   7.0711   9.1924

s = std(X,0,2)
s =
   4.000
   7.5056
```

See Also

corrcoef, cov, mean, median

Purpose

Plot discrete sequence data

Syntax

```
stem(Y)
stem(X,Y)
stem(...,'fill')
stem(...,LineSpec)
h = stem(...)
```

Description

A two-dimensional stem plot displays data as lines extending from the *x*-axis. A circle (the default) or other marker whose *y*-position represents the data value terminates each stem.

`stem(Y)` plots the data sequence `Y` as stems that extend from equally spaced and automatically generated values along the *x*-axis. When `Y` is a matrix, `stem` plots all elements in a row against the same *x* value.

`stem(X,Y)` plots `X` versus the columns of `Y`. `X` and `Y` are vectors or matrices of the same size. Additionally, `X` can be a row or a column vector and `Y` a matrix with `length(X)` rows.

`stem(...,'fill')` specifies whether to color the circle at the end of the stem.

`stem(...,LineSpec)` specifies the line style, marker symbol, and color for the stem plot. See `LineSpec` for more information.

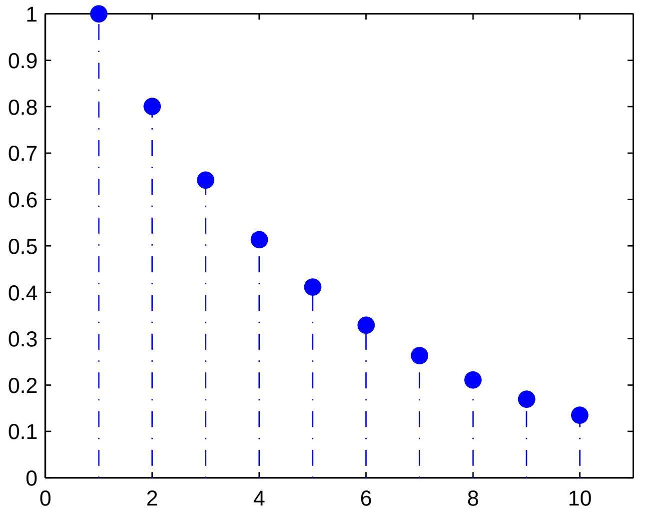
`h = stem(...)` returns handles to line graphics objects.

Examples

Create a stem plot of 10 random numbers.

```
y = linspace(0,2,10);
stem(exp(-y),'fill','–.')
```

```
axis ([0 11 0 1])
```



See Also

`bar`, `plot`, `stairs`, `stem3`

Purpose

Plot three-dimensional discrete sequence data

Syntax

```
stem3(Z)
stem3(X,Y,Z)
stem3(...,'fill')
stem3(...,LineSpec)
h = stem3(...)
```

Description

Three-dimensional stem plots display lines extending from the *xy*-plane. A circle (the default) or other marker symbol whose *z*-position represents the data value terminates each stem.

`stem3(Z)` plots the data sequence `Z` as stems that extend from the *xy*-plane. *x* and *y* are generated automatically. When `Z` is a row vector, `stem3` plots all elements at equally spaced *x* values against the same *y* value. When `Z` is a column vector, `stem3` plots all elements at equally spaced *y* values against the same *x* value.

`stem3(X,Y,Z)` plots the data sequence `Z` at values specified by `X` and `Y`. `X`, `Y`, and `Z` must all be vectors or matrices of the same size.

`stem3(...,'fill')` specifies whether to color the interior of the circle at the end of the stem.

`stem3(...,LineSpec)` specifies the line style, marker symbol, and color for the stems. See `LineSpec` for more information.

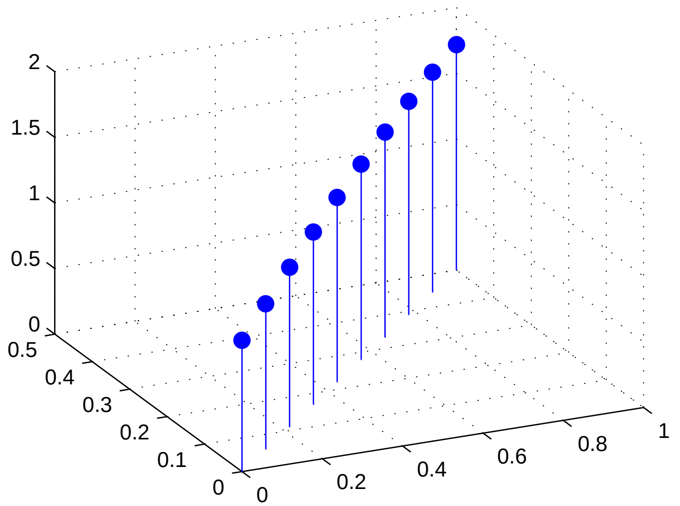
`h = stem3(...)` returns handles to line graphics objects.

Examples

Create a three-dimensional stem plot to visualize a function of two variables.

```
X = linspace(0,1,10);
Y = X./2;
Z = sin(X) + cos(Y);
stem3(X,Y,Z,'fill')
view(-25,30)
```

:



See Also

`bar`, `plot`, `stairs`, `stem`

Purpose Stop asynchronous read and write operations

Syntax `stopasync(obj)`

Arguments

`obj`	A serial port object or an array of serial port objects.

Description `stopasync(obj)` stops any asynchronous read or write operation that is in progress for `obj`.

Remarks You can write data asynchronously using the `fprintf` or `fwrite` functions. You can read data asynchronously using the `readasync` function, or by configuring the `ReadAsyncMode` property to `continuous`. In-progress asynchronous operations are indicated by the `TransferStatus` property.

If `obj` is an array of serial port objects and one of the objects cannot be stopped, the remaining objects in the array are stopped and a warning is returned. After an object stops:

- Its `TransferStatus` property is configured to `idle`.
- Its `ReadAsyncMode` property is configured to `manual`.
- The data in its output buffer is flushed.

Data in the input buffer is not flushed. You can return this data to the MATLAB workspace using any of the synchronous read functions. If you execute the `readasync` function, or configure the `ReadAsyncMode` property to `continuous`, then the new data is appended to the existing data in the input buffer.

See Also

Functions

`fprintf`, `fwrite`, `readasync`

Properties

`ReadAsyncMode`, `TransferStatus`

str2double

Purpose Convert string to double-precision value

Syntax

```
x = str2double('str')
X = str2double(C)
```

Description `X = str2double('str')` converts the string *str*, which should be an ASCII character representation of a real or complex scalar value, to MATLAB's double-precision representation. The string may contain digits, a comma (thousands separator), a decimal point, a leading + or - sign, an `e` preceeding a power of 10 scale factor, and an `i` for a complex unit.

If *str* does not represent a valid scalar value, `str2double` returns `NaN`.

`X = str2double(C)` converts the strings in the cell array of strings `C` to double-precision. The matrix `X` returned will be the same size as `C`.

Examples Here are some valid `str2double` conversions.

```
str2double('123.45e7')
str2double('123 + 45i')
str2double('3.14159')
str2double('2.7i - 3.14')
str2double({'2.71' '3.1415'})
str2double('1,200.34')
```

See Also `char`, `hex2num`, `num2str`, `str2num`

Purpose Constructs a function handle from a function name string

Syntax

```
fhandle = str2func('str')
```

Description `str2func('str')` constructs a function handle, `fhandle`, for the function named in the string, `'str'`.

You can create a function handle using either the `@function` syntax or the `str2func` command. You can also perform this operation on a cell array of strings. In this case, an array of function handles is returned.

Examples To create a function handle from the function name, `'humps'`

```
fhandle = str2func('humps')

fhandle =

    @humps
```

To create an array of function handles from a cell array of function names

```
fh_array = str2func({'sin' 'cos' 'tan'})

fh_array =

    @sin    @cos    @tan
```

See Also `function_handle`, `func2str`, `functions`

str2mat

Purpose

Form a blank padded character matrix from strings

Syntax

```
S = str2mat(T1,T2,T3,..)
```

Description

`S = str2mat(T1,T2,T3,..)` forms the matrix S containing the text strings `T1,T2,T3,...` as rows. The function automatically pads each string with blanks in order to form a valid matrix. Each text parameter, Ti, can itself be a string matrix. This allows the creation of arbitrarily large string matrices. Empty strings are significant.

Note This routine will become obsolete in a future version. Use `char` instead.

Remarks

`str2mat` differs from `strvcat` in that empty strings produce blank rows in the output. In `strvcat`, empty strings are ignored.

Examples

```
x = str2mat('36842','39751','38453','90307');

whos x
  Name      Size         Bytes  Class

  x         4x5             40  char array

x(2,3)

ans =

    7
```

See Also

`char`, `strvcat`

Purpose String to number conversion

Syntax

```
x = str2num('str')
```

Description

`x = str2num('str')` converts the string *str*, which is an ASCII character representation of a numeric value, to MATLAB's numeric representation. The string can contain:

- Digits
- A decimal point
- A leading + or - sign
- A letter `e` or `d` preceding a power of 10 scale factor
- A letter `i` or `j` indicating a complex or imaginary number.

The `str2num` function can also convert string matrices.

Examples

`str2num('3.14159e0')` is approximately π.

To convert a string matrix:

```
str2num(['1 2';'3 4'])

ans =

     1     2
     3     4
```

See Also

`num2str`, `hex2num`, `sscanf`, `sparse`, `special characters`

Purpose String concatenation

Syntax

```
t = strcat(s1,s2,s3,...)
```

Description

`t = strcat(s1,s2,s3,...)` horizontally concatenates corresponding rows of the character arrays `s1`, `s2`, `s3`, etc. All input arrays must have the same number of rows (or any can be a single string). When the inputs are all character arrays, the output is also a character array.

When any of the inputs is a cell array of strings, `strcat` returns a cell array of strings formed by concatenating corresponding elements of `s1`, `s2`, etc. The inputs must all have the same size (or any can be a scalar). Any of the inputs can also be character arrays.

Trailing spaces in character array inputs are ignored and do not appear in the output. This is not true for inputs that are cell arrays of strings. Use the concatenation syntax `[s1 s2 s3 ...]` to preserve trailing spaces.

Remarks

`strcat` and matrix operation are different for strings that contain trailing spaces:

```
a = 'hello '
b = 'goodbye'
strcat(a,b)
ans =
hellogoodbye
[a b]
ans =
hello goodbye
```

Examples

Given two 1-by-2 cell arrays `a` and `b`,

```
a =                           b =
    'abcde'    'fghi'             'jkl'    'mn'
```

the command `t = strcat(a,b)` yields:

```
t =
    'abcdejkl'    'fghimn'
```

Given the 1-by-1 cell array `c = {'Q'}`, the command `t = strcat(a,b,c)` yields:

```
t =
    'abcdejklQ'    'fghimnQ'
```

See Also strvcat, cat, cellstr

strcmp

Purpose Compare strings

Syntax

```
k = strcmp('str1','str2')
TF = strcmp(S,T)
```

Description `k = strcmp('`*`str1`*`','`*`str2`*`')` compares the strings *`str1`* and `str2` and returns logical true (1) if the two are identical, and logical false (`0`) otherwise.

`TF = strcmp(S,T)` where either `S` or `T` is a cell array of strings, returns an array `TF` the same size as `S` and `T` containing `1` for those elements of `S` and `T` that match, and `0` otherwise. `S` and `T` must be the same size (or one can be a scalar cell). Either one can also be a character array with the right number of rows.

Remarks Note that the value returned by `strcmp` is not the same as the C language convention. In addition, the `strcmp` function is case sensitive; any leading and trailing blanks in either of the strings are explicitly included in the comparison.

Examples

```
strcmp('Yes','No') =
   0
strcmp('Yes','Yes') =
   1

A =
    'MATLAB'              'SIMULINK'
    'Toolboxes'           'The MathWorks'

B =
    'Handle Graphics'     'Real Time Workshop'
    'Toolboxes'           'The MathWorks'

C =
    'Signal Processing'     'Image Processing'
    'MATLAB'                'SIMULINK'

strcmp(A,B)
ans =
     0     0
     1     1

strcmp(A,C)
```

```
ans =
     0     0
     0     0
```

See Also strncmp, strcmpi, strncmpi, strmatch, findstr

strcmpi

Purpose Compare strings ignoring case

Syntax

```
strcmpi(str1,str2)
strcmpi(S,T)
```

Description `strcmpi(`*`str1`*`,`*`str2`*`)` returns 1 if strings *`str1`* and *`str2`* are the same except for case and 0 otherwise.

`strcmpi(S,T)` when either `S` or `T` is a cell array of strings, returns an array the same size as `S` and `T` containing 1 for those elements of `S` and `T` that match except for case, and 0 otherwise. `S` and `T` must be the same size (or one can be a scalar cell). Either one can also be a character array with the right number of rows.

`strcmpi` supports international character sets.

See Also `findstr`, `strcmp`, `strmatch`, `strncmpi`

Purpose Compute 2-D stream line data

Syntax

```
XY = stream2(x,y,u,v,startx,starty)
XY = stream2(u,v,startx,starty)
XY = stream2(...,options)
```

Description `XY = stream2(x,y,u,v,startx,starty)` computes stream lines from vector data `u` and `v`. The arrays `x` and `y` define the coordinates for `u` and `v` and must be monotonic and 2-D plaid (such as the data produced by `meshgrid`). `startx` and `starty` define the starting positions of the stream lines. The section "Starting Points for Stream Plots" in Visualization Techniques provides more information on defining starting points.

The returned value `XY` contains a cell array of vertex arrays.

`XY = stream2(u,v,startx,starty)` assumes the arrays `x` and `y` are defined as `[x,y] = meshgrid(1:n,1:m)` where `[m,n] = size(u)`.

`XY = stream2(...,options)` specifies the options used when creating the stream lines. Define `options` as a one or two element vector containing the step size or the step size and the maximum number of vertices in a stream line:

```
[stepsize]
```

or

```
[stepsize, max_number_vertices]
```

If you do not specify a value, MATLAB uses the default:

- stepsize = 0.1 (one tenth of a cell)
- naximum number of vertices = 1000

Use the `streamline` command to plot the data returned by `stream2`.

Examples This example draws 2-D stream lines from data representing air currents over regions of North America.

```
load wind
[sx,sy] = meshgrid(80,20:10:50);
streamline(stream2(x(:,:,5),y(:,:,5),u(:,:,5),v(:,:,5),sx,sy));
```

See Also

`coneplot`, `isosurface`, `reducevolume` `smooth3`, `stream3`, `streamline`, `subvolume`

Purpose

Compute 3-D stream line data

Syntax

```
XYZ = stream3(X,Y,Z,U,V,W,startx,starty,startz)
XYZ = stream3(U,V,W,startx,starty,startz)
```

Description

`XYZ = stream3(X,Y,Z,U,V,W,startx,starty,startz)` computes stream lines from vector data `U`, `V`, `W`. The arrays `X`, `Y`, `Z` define the coordinates for `U`, `V`, `W` and must be monotonic and 3-D plaid (such as the data produced by `meshgrid`). `startx`, `starty`, and `startz` define the starting positions of the stream lines. The section "Starting Points for Stream Plots" in Visualization Techniques provides more information on defining starting points.

The returned value `XYZ` contains a cell array of vertex arrays.

`XYZ = stream3(U,V,W,startx,starty,startz)` assumes the arrays `X`, `Y`, and `Z` are defined as `[X,Y,Z] = meshgrid(1:N,1:M,1:P)` where `[M,N,P] = size(U)`.

`XYZ = stream3(...,options)` specifies the options used when creating the stream lines. Define `options` as a one or two element vector containing the step size or the step size and the maximum number of vertices in a stream line:

```
[stepsize]
```

or

```
[stepsize, max_number_vertices]
```

If you do not specify values, MATLAB uses the default:

- stepsize = 0.1 (one tenth of a cell)
- naximum number of vertices = 1000

Use the `streamline` command to plot the data returned by `stream3`.

Examples

This example draws 3-D stream lines from data representing air currents over regions of North America.

```
load wind
[sx sy sz] = meshgrid(80,20:10:50,0:5:15);
streamline(stream3(x,y,z,u,v,w,sx,sy,sz))
view(3)
```

See Also

coneplot, isosurface, reducevolume smooth3, stream2, streamline, subvolume

Purpose Draw stream lines from 2-D or 3-D vector data

Syntax

```
h = streamline(X,Y,Z,U,V,W,startx,starty,startz)
h = streamline(U,V,W,startx,starty,startz)
h = streamline(XYZ)
h = streamline(X,Y,U,V,startx,starty)
h = streamline(U,V,startx,starty)
h = streamline(XY)
h = streamline(...,options)
```

Description

`h = streamline(X,Y,Z,U,V,W,startx,starty,startz)` draws stream lines from 3-D vector data `U`, `V`, `W`. The arrays `X`, `Y`, `Z` define the coordinates for `U`, `V`, `W` and must be monotonic and 3-D plaid (such as the data produced by `meshgrid`). `startx`, `starty`, `startz` define the starting positions of the stream lines. The section "Starting Points for Stream Plots" in Visualization Techniques provides more information on defining starting points.

The output argument `h` contains a vector of line handles, one handle for each stream line.

`h = streamline(U,V,W,startx,starty,startz)` assumes the arrays `X`, `Y`, and `Z` are defined as `[X,Y,Z] = meshgrid(1:N,1:M,1:P)` where `[M,N,P] = size(U)`.

`h = streamline(XYZ)` assumes `XYZ` is a precomputed cell array of vertex arrays (as produced by `stream3`).

`h = streamline(X,Y,U,V,startx,starty)` draws stream lines from 2-D vector data `U`, `V`. The arrays `X`, `Y` define the coordinates for `U`, `V` and must be monotonic and 2-D plaid (such as the data produced by `meshgrid`). `startx` and `starty` define the starting positions of the stream lines. The output argument `h` contains a vector of line handles, one handle for each stream line.

`h = streamline(U,V,startx,starty)` assumes the arrays `X` and `Y` are defined as `[X,Y] = meshgrid(1:N,1:M)` where `[M,N] = size(U)`.

`h = streamline(XY)` assumes `XY` is a precomputed cell array of vertex arrays (as produced by `stream2`).

`streamline(...,options)` specifies the options used when creating the stream lines. Define `options` as a one or two element vector containing the step size or the step size and the maximum number of vertices in a stream line:

```
[stepsize]
```

or

```
[stepsize, max_number_vertices]
```

If you do not specify values, MATLAB uses the default:

- stepsize = 0.1 (one tenth of a cell)
- naximum number of vertices = 1000

Examples

This example draws stream lines from data representing air currents over a region of North America. Loading the `wind` data set creates the variables `x`, `y`, `z`, `u`, `v`, and `w` in the MATLAB workspace.

The plane of stream lines indicates the flow of air from the west to the east (the x direction) beginning at x = 80 (which is close to the minimum value of the x coordinates). The y and z coordinate starting points are multivalued and approximately span the range of these coordinates. `meshgrid` generates the starting positions of the stream lines.

```
load wind
[sx,sy,sz] = meshgrid(80,20:10:50,0:5:15);
h = streamline(x,y,z,u,v,w,sx,sy,sz);
set(h,'Color','red')
view(3)
```

See Also

`stream2`, `stream3`, `coneplot`, `isosurface`, `smooth3`, `subvolume`, `reducevolume`

Purpose

Display stream particles

Syntax

```
streamparticles(vertices)
streamparticles(vertices,n)
streamparticles(...,'PropertyName',PropertyValue,...)
streamparticles(line_handle,...)
h = streamparticles(...)
```

Description

`streamparticles(vertices)` draws stream particles of a vector field. Stream particles are usually represented by markers and can show the position and velocity of a streamline. `vertices` is a cell array of 2-D or 3-D vertices (as if produced by `stream2` or `stream3`).

`streamparticles(vertices,n)` uses `n` to determine how many stream particles to draw. The `ParticleAlignment` property controls how `n` is interpreted.

- If `ParticleAlignment` is set to `off` (the default) and `n` is greater than `1`, then approximately `n` particles are drawn evenly spaced over the streamline vertices.

  If `n` is less than or equal to `1`, `n` is interpreted as a fraction of the original stream vertices; for example, if `n` is `0.2`, approximately 20% of the vertices are used.

  `n` determines the upper bound for the number of particles drawn. Note that the actual number of particles may deviate from `n` by as much as a factor of 2.

- If `ParticleAlignment` is `on`, `n` determines the number of particles on the streamline having the most vertices and sets the spacing on the other streamlines to this value. The default value is `n = 1`.

`streamparticles(...,'`*`PropertyName`*`',PropertyValue,...)` controls the stream particles using named properties and specified values. Any unspecified properties have default values. MATLAB ignores the case of property names.

Stream Particle Properties

`Animate` – Stream particle motion [non-negative integer]

The number of times to animate the stream particles. The default is `0`, which does not animate. `Inf` animates until you enter **ctrl-c**.

FrameRate – Animation frames per second [non-negative integer]

This property specifies the number of frames per second for the animation. Inf, the default draws the animation as fast as possible. Note that speed of the animation may be limited by the speed of the computer. In such cases, the value of FrameRate can not necessarily be achieved.

ParticleAlignment – Align particles with stream lines [on | {off}]

Set this property to on to draw particles at the beginning of each the stream line. This property controls how streamparticles interprets the argument n (number of stream particles).

Stream particles are line objects. In addition to stream particle properties, you can specify any line object property, such as Marker and EraseMode. streamparticles sets the following line properties when called.

Line Property	Value Set by streamparticles
EraseMode	xor
LineStyle	none
Marker	o
MarkerEdgeColor	none
MarkerFaceColor	red

You can override any of these properties by specifying a property name and value as arguments to streamparticles. For example, this statement uses RGB values to set the MarkerFaceColor to medium gray:

```
streamparticles(vertices,'MarkerFaceColor',[.5 .5 .5])
```

streamparticles(line_handle,...) uses the line object identified by line_handle to draw the stream particles.

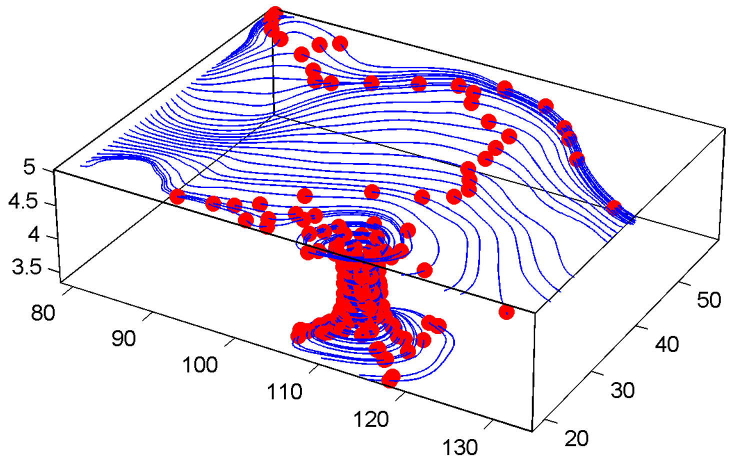
h = streamparticles(...) returns a vector of handles to the line objects it creates.

Examples

This example combines stream lines with stream particle animation. The interpstreamspeed function determines the vertices along the stream lines where stream particles will be drawn during the animation, thereby controlling the speed of the animation. Setting the axes `DrawMode` property to `fast` provides faster rendering.

```
load wind
[sx sy sz] = meshgrid(80,20:1:55,5);
verts = stream3(x,y,z,u,v,w,sx,sy,sz);
sl = streamline(verts);
iverts = interpstreamspeed(x,y,z,u,v,w,verts,.025);
axis tight; view(30,30); daspect([1 1 .125])
camproj perspective; camva(8)
set(gca,'DrawMode','fast')
box on
streamparticles(iverts,35,'animate',10,'ParticleAlignment','on'
)
```

The following picture is a static view of the animation.



This example uses the stream lines in the `z = 5` plane to animate the flow along these lines with steamparticles.

```
load wind
daspect([1 1 1]); view(2)
[verts averts] = streamslice(x,y,z,u,v,w,[],[],[5]);
```

```
sl = streamline([verts averts]);
axis tight off;
set(sl,'Visible','off')
iverts = interpstreamspeed(x,y,z,u,v,w,verts,.05);
set(gca,'DrawMode','fast','Position',[0 0 1 1],'ZLim',[4.9 5.1])
set(gcf,'Color','black')
streamparticles(iverts, 200, ...
    'Animate',100,'FrameRate',40, ...
    'MarkerSize',10,'MarkerFaceColor','yellow')
```

See Also

isosurface, isocaps, smooth3, subvolume, reducevolume, reducepatch, isonormals

Purpose Creates a 3-D stream ribbon plot

Syntax

```
streamribbon(X,Y,Z,U,V,W,startx,starty,startz)
streamribbon(U,V,W,startx,starty,startz)
streamribbon(vertices,X,Y,Z,cav,speed)
streamribbon(vertices,cav,speed)
streamribbon(vertices,twistangle)
streamribbon(...,width)
h = streamribbon(...)
```

Description

streamribbon(X,Y,Z,U,V,W,startx,starty,startz) draws stream ribbons from vector volume data U, V, W. The arrays X, Y, Z define the coordinates for U, V, W and must be monotonic and 3-D plaid (as if produced by meshgrid). startx, starty, and startz define the starting positions of the stream ribbons at the center of the ribbons. The section "Starting Points for Stream Plots" in Visualization Techniques provides more information on defining starting points.

The twist of the ribbons is proportional to the curl of the vector field. The width of the ribbons is calculated automatically.

Generally, you should set the DataAspectRatio (daspect) before calling streamribbon.

streamribbon(U,V,W,startx,starty,startz) assumes X, Y, and Z are determined by the expression:

```
[X,Y,Z] = meshgrid(1:n,1:m,1:p)
```

where [m,n,p] = size(U).

streamribbon(vertices,X,Y,Z,cav,speed) assumes precomputed streamline vertices, curl angular velocity, and flow speed. vertices is a cell array of stream line vertices (as produced by stream3). X, Y, Z, cav, and speed are 3-D arrays.

streamribbon(vertices,cav,speed) assumes X, Y, and Z are determined by the expression:

```
[X,Y,Z] = meshgrid(1:n,1:m,1:p)
```

where [m,n,p] = size(cav)

streamribbon(vertices,twistangle) uses the cell array of vectors twistangle for the twist of the ribbons (in radians). The size of each corresponding element of vertices and twistangle must be equal.

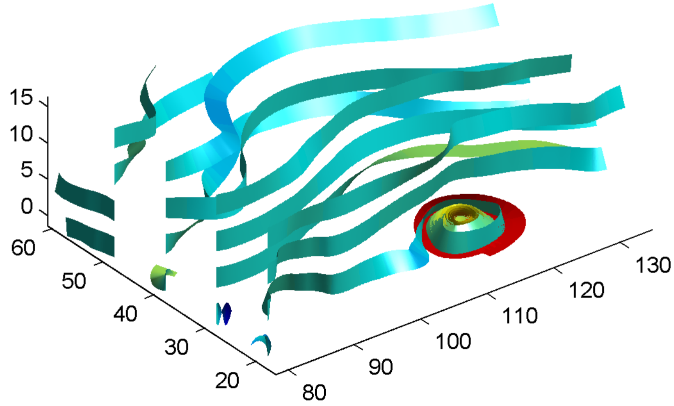
streamribbon(...,width) sets the width of the ribbons to width.

h = streamribbon(...) returns a vector of handles (one per start point) to surface objects.

Examples

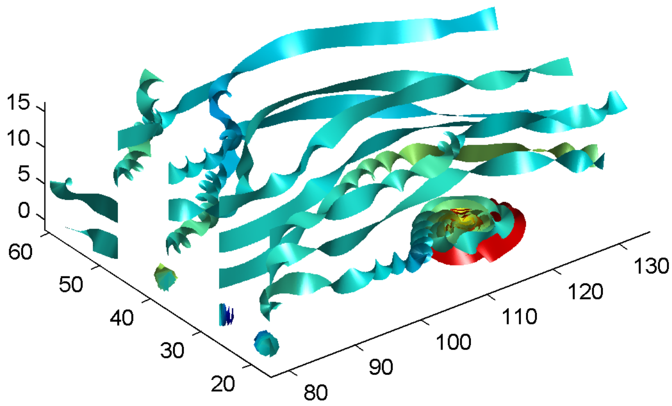
This example uses stream ribbons to indicate the flow in the wind data set. Inputs include the coordinates, vector field components, and starting location for the stream ribbons.

```
load wind
[sx sy sz] = meshgrid(80,20:10:50,0:5:15);
daspect([1 1 1])
streamribbon(x,y,z,u,v,w,sx,sy,sz);
%-----Define viewing and lighting
axis tight
shading interp;
view(3);
camlight; lighting gouraud
```



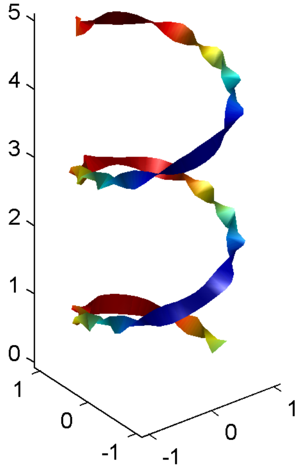
This example uses precalculated vertex data (`stream3`), curl average velocity (`curl`), and speed ($\sqrt{u^2 + v^2 + w^2}$). Using precalculated data enables you to use values other than those calculated from the single data source. In this case, the speed is reduced by a factor of 10 compared to the previous example.

```
load wind
[sx sy sz] = meshgrid(80,20:10:50,0:5:15);
daspect([1 1 1])
verts = stream3(x,y,z,u,v,w,sx,sy,sz);
cav = curl(x,y,z,u,v,w);
spd = sqrt(u.^2 + v.^2 + w.^2).*.1;
streamribbon(verts,x,y,z,cav,spd);
%-----Define viewing and lighting
axis tight
shading interp
view(3)
camlight; lighting gouraud
```



This example specifies a twist angle for the stream ribbon.

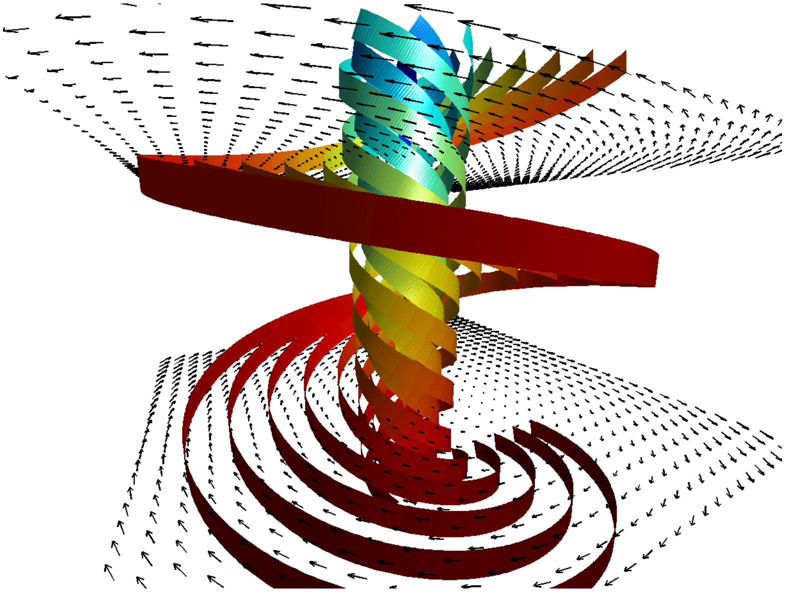
```
t = 0:.15:15;
verts = {[cos(t)' sin(t)' (t/3)']};
twistangle = {cos(t)'};
daspect([1 1 1])
streamribbon(verts,twistangle);
%-----Define viewing and lighting
axis tight
shading interp;
view(3);
camlight; lighting gouraud
```



This example combines cone plots (`coneplot`) and stream ribbon plots in one graph.

```
%-----Define 3-D arrays x, y, z, u, v, w
xmin = -7; xmax = 7;
ymin = -7; ymax = 7;
zmin = -7; zmax = 7;
x = linspace(xmin,xmax,30);
y = linspace(ymin,ymax,20);
z = linspace(zmin,zmax,20);
[x y z] = meshgrid(x,y,z);
u = y; v = -x; w = 0*x+1;
daspect([1 1 1]);
[cx cy cz] = meshgrid(linspace(xmin,xmax,30),...
   linspace(ymin,ymax,30),[-3 4]);
h = coneplot(x,y,z,u,v,w,cx,cy,cz,'quiver');
set(h,'color','k');
%-----Plot two sets of streamribbons
[sx sy sz] = meshgrid([-1 0 1],[-1 0 1],-6);
streamribbon(x,y,z,u,v,w,sx,sy,sz);
[sx sy sz] = meshgrid([1:6],[0],-6);
streamribbon(x,y,z,u,v,w,sx,sy,sz);
```

```
%-----Define viewing and lighting
shading interp
view(-30,10) ; axis off tight
camproj perspective; camva(66); camlookat;
camdolly(0,0,.5,'fixtarget')
camlight
```



See also `curl`, `streamtube`, `streamline`, `stream3`

Purpose Draws stream lines in slice planes

Syntax

```
streamslice(X,Y,Z,U,V,W,startx,starty,startz)
streamslice(U,V,W,startx,starty,startz)
streamslice(X,Y,U,V)
streamslice(U,V)
streamslice(...,density)
streamslice(...,'arrowmode')
streamslice(...,'method')
h = streamslice(...)
[vertices arrowvertices] = streamslice(...)
```

Description

`streamslice(X,Y,Z,U,V,W,startx,starty,startz)` draws well spaced streamlines (with direction arrows) from vector data `U`, `V`, `W` in axis aligned x-, y-, z-planes at the points in the vectors `startx`, `starty`, `startz`. (The section "Starting Points for Stream Plots" in Visualization Techniques provides more information on defining starting points.) The arrays `X`, `Y`, `Z` define the coordinates for `U`, `V`, `W` and must be monotonic and 3-D plaid (as if produced by `meshgrid`). `U`, `V`, `W` must be m-by-n-by-p volume arrays.

You should not assumed that the flow is parallel to the slice plane. For example, in a stream slice at a constant `z`, the `z` component of the vector field, `W`, is ignored when calculating the streamlines for that plane.

Stream slices are useful for determining where to start stream lines, stream tubes, and stream ribbons.

`streamslice(U,V,W,startx,starty,startz)` assumes `X`, `Y`, and `Z` are determined by the expression:

```
[X,Y,Z] = meshgrid(1:n,1:m,1:p)
```

where `[m,n,p] = size(U)`.

`streamslice(X,Y,U,V)` draws well spaced stream lines (with direction arrows) from vector volume data `U`, `V`. The arrays `X`, `Y` define the coordinates for `U`, `V` and must be monotonic and 2-D plaid (as if produced by `meshgrid`).

`streamslice(U,V)` assumes `X`, `Y`, and `Z` are determined by the expression:

```
[X,Y,Z] = meshgrid(1:n,1:m,1:p)
```

where [m,n,p] = size(U)

streamslice(...,density) modifies the automatic spacing of the stream lines. density must be greater than 0. The default value is 1; higher values produce more stream lines on each plane. For example, 2 produces approximately twice as many stream lines, while 0.5 produces approximately half as many.

streamslice(...,'*arrowsmode*') determines if direction arrows are present or not. arrowmode can be:

- arrows – draw direction arrows on the streamlines (default)
- noarrows – does not draw direction arrows

streamslice(...,'*method*') specifies the interpolation method to use. method can be:

- linear – linear interpolation (default)
- cubic – cubic interpolation
- nearest – nearest neighbor interpolation

See interp3 for more information interpolation methods.

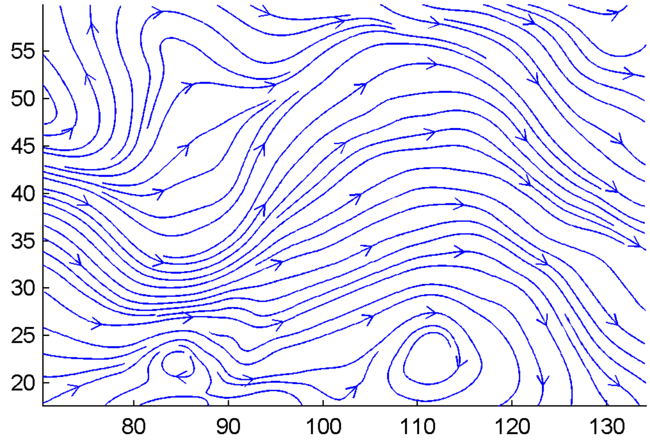
h = streamslice(...) returns a vector of handles to the line objects created.

[vertices arrowvertices] = streamslice(...) returns two cell arrays of vertices for drawing the stream lines and the arrows. You can pass these values to any of the stream line drawing functions (streamline, streamribbon, streamtube)

Examples

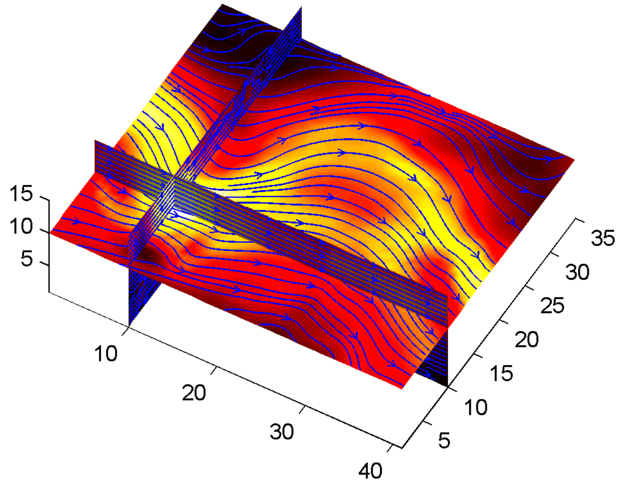
This example creates a stream slice in the wind data set at z = 5.

```
load wind
daspect([1 1 1])
streamslice(x,y,z,u,v,w,[],[],[5])
axis tight
```



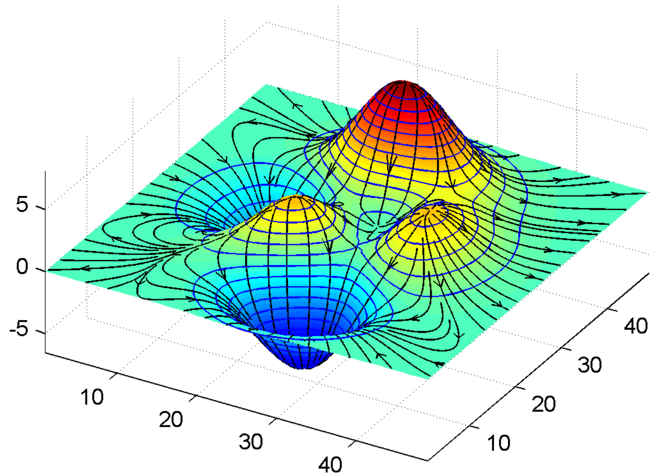
This example uses `streamslice` to calculate vertex data for the stream lines and the direction arrows. This data is then used by `streamline` to plot the lines and arrows. Slice planes illustrating with color the wind speed ($\sqrt{u^2 + v^2 + w^2}$) are drawn by slice in the same planes.

```
load wind
daspect([1 1 1])
[verts averts] = streamslice(u,v,w,10,10,10);
streamline([verts averts])
spd = sqrt(u.^2 + v.^2 + w.^2);
hold on;
slice(spd,10,10,10);
colormap(hot)
shading interp
view(30,50); axis(volumebounds(spd));
camlight; material([.5 1 0])
```



This example superimposes contour lines on a surface and then uses `streamslice` to draw lines that indicate the gradient of the surface. `interp2` is used to find the points for the lines that lie on the surface.

```
z = peaks;
surf(z)
shading interp
hold on
[c ch] = contour3(z,20); set(ch,'edgecolor','b')
[u v] = gradient(z);
h = streamslice(-u,-v);
set(h,'color','k')
for i=1:length(h);
   zi = interp2(z,get(h(i),'xdata'),get(h(i),'ydata'));
   set(h(i),'zdata',zi);
end
view(30,50); axis tight
```



See also `contourslice`, `slice`, `streamline`, `volumebounds`

streamtube

Purpose

Creates a 3-D stream tube plot

Syntax

```
streamtube(X,Y,Z,U,V,W,startx,starty,startz)
streamtube(U,V,W,startx,starty,startz)
streamtube(vertices,X,Y,Z,divergence)
streamtube(vertices,divergence)
streamtube(vertices,width)
streamtube(vertices)
streamtube(...,[scale n])
h = streamtube(...)
```

Description

`streamtube(X,Y,Z,U,V,W,startx,starty,startz)` draws stream tubes from vector volume data `U`, `V`, `W`. The arrays `X`, `Y`, `Z` define the coordinates for `U`, `V`, `W` and must be monotonic and 3-D plaid (as if produced by `meshgrid`). `startx`, `starty`, and `startz` define the starting positions of the stream lines at the center of the tubes. The section "Starting Points for Stream Plots" in Visualization Techniques provides more information on defining starting points.

The width of the tubes is proportional to the normalized divergence of the vector field.

Generally, you should set the `DataAspectRatio` (`daspect`) before calling `streamtube`.

`streamtube(U,V,W,startx,starty,startz)` assumes `X`, `Y`, and `Z` are determined by the expression:

```
[X,Y,Z] = meshgrid(1:n,1:m,1:p)
```

where `[m,n,p] = size(U)`.

`streamtube(vertices,X,Y,Z,divergence)` assumes precomputed stream line vertices and divergence. `vertices` is a cell array of stream line vertices (as produced by `stream3`). `X`, `Y`, `Z`, and `divergence` are 3-D arrays.

`streamtube(vertices,divergence)` assumes `X`, `Y`, and `Z` are determined by the expression:

```
[X,Y,Z] = meshgrid(1:n,1:m,1:p)
```

where [m,n,p] = size(divergence)

streamtube(vertices,width) specifies the width of the tubes in the cell array of vectors, width. The size of each corresponding element of vertices and width must be equal. width can also be a scalar, specifying a single value for the width of all stream tubes.

streamtube(vertices) selects the width automatically.

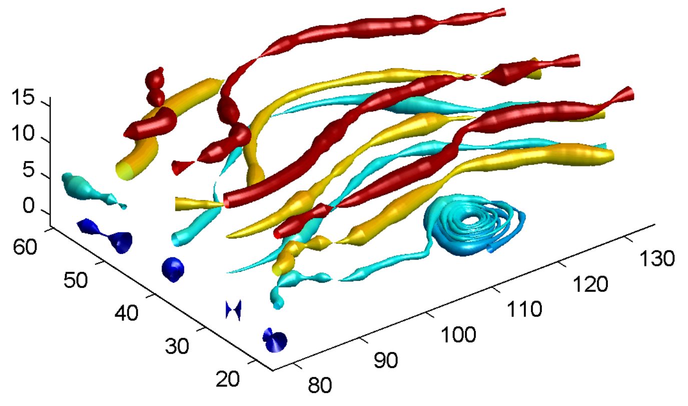
streamtube(...,[scale n]) scales the width of the tubes by scale. The default is scale = 1. When the stream tubes are created using start points or divergence, specifying scale = 0 suppresses automatic scaling. n is the number of points along the circumference of the tube. The default is n = 20.

h = streamtube(...z) returns a vector of handles (one per start point) to surface objects used to draw the stream tubes.

Examples

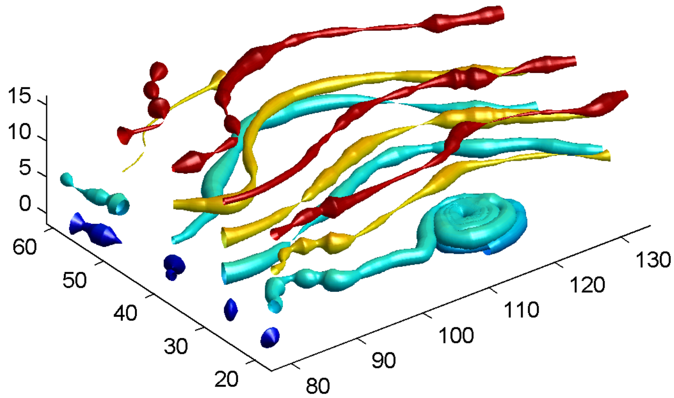
This example uses stream tubes to indicate the flow in the wind data set. Inputs include the coordinates, vector field components, and starting location for the stream tubes.

```
load wind
[sx sy sz] = meshgrid(80,20:10:50,0:5:15);
daspect([1 1 1])
streamtube(x,y,z,u,v,w,sx,sy,sz);
%-----Define viewing and lighting
view(3)
axis tight
shading interp;
camlight; lighting gouraud
```



This example uses precalculated vertex data (`stream3`) and divergence (`divergence`).

```
load wind
[sx sy sz] = meshgrid(80,20:10:50,0:5:15);
daspect([1 1 1])
verts = stream3(x,y,z,u,v,w,sx,sy,sz);
div = divergence(x,y,z,u,v,w);
streamtube(verts,x,y,z,-div);
%-----Define viewing and lighting
view(3)
axis tight
shading interp
camlight; lighting gouraud
```



See also divergence, streamribbon, streamline, stream3

strfind

Purpose Find one string within another

Syntax

```
k = strfind(str,pattern)
```

Description `k = strfind(str,pattern)` searches the string, `str`, for occurrences of a shorter string, `pattern`, returning the starting index of each such occurrence in the double array, `k`. If `pattern` is not found in `str`, or if `pattern` is longer than `str`, then `strfind` returns the empty array, `[]`.

The search performed by `strfind` is case sensitive. Any leading and trailing blanks in either `str` or `pattern` are explicitly included in the comparison.

Use the function `findstr`, if you are not certain which of the two input strings is the longer one.

Examples

```
s = 'Find the starting indices of the pattern string';
strfind(s,'in')
ans =
     2    15    19    45

strfind(s,'In')
ans =
     []

strfind(s,' ')
ans =
     5     9    18    26    29    33    41
```

See Also `findstr`, `strmatch`, `strtok`, `strcmp`, `strncmp`, `strcmpi`, `strncmpi`

Purpose MATLAB string handling

Syntax

```
S = 'Any Characters'
S = char(X)
X = double(S)
```

Description

`S = 'Any Characters'` creates a character array, or string. The string is actually a vector whose components are the numeric codes for the characters (the first 127 codes are ASCII). The actual characters displayed depend on the character set encoding for a given font. The length of `S` is the number of characters. A quote within the string is indicated by two quotes.

`S = [S1 S2 ...]` concatenates character arrays `S1`, `S2`, etc. into a new character array, `S`.

`S = strcat(S1, S2, ...)` concatenates `S1`, `S2`, etc., which can be character arrays or cell arrays of strings. When the inputs are all character arrays, the output is also a character array. When any of the inputs is a cell array of strings, `strcat` returns a cell array of strings.

Trailing spaces in `strcat` character array inputs are ignored and do not appear in the output. This is not true for `strcat` inputs that are cell arrays of strings. Use the `S = [S1 S2 ...]` concatenation syntax, shown above, to preserve trailing spaces.

`S = char(X)` can be used to convert an array that contains positive integers representing numeric codes into a MATLAB character array.

`X = double(S)` converts the string to its equivalent double precision numeric codes.

A collection of strings can be created in either of the following two ways:

- As the rows of a character array via `strvcat`
- As a cell array of strings via the curly braces

You can convert between character array and cell array of strings using `char` and `cellstr`. Most string functions support both types.

`ischar(S)` tells if `S` is a string variable. `iscellstr(S)` tells if `S` is a cell array of strings.

Examples

Create a simple string that includes a single quote.

```
msg = 'You''re right!'

msg =
You're right!
```

Create the string, name, using two methods of concatenation.

```
name = ['Thomas' ' R. ' 'Lee']
```

```
name = strcat('Thomas',' R.',' Lee')
```

Create a vertical array of strings.

```
C = strvcat('Hello','Yes','No','Goodbye')

C =
Hello
Yes
No
Goodbye
```

Create a cell array of strings.

```
S = {'Hello' 'Yes' 'No' 'Goodbye'}

S =
    'Hello'    'Yes'    'No'    'Goodbye'
```

See Also

`char`, `cellstr`, `ischar`, `iscellstr`, `strvcat`, `sprintf`, `sscanf`, `input`

Purpose Justify a character array

Syntax

```
T = strjust(S)
T = strjust(S,'right')
T = strjust(S,'left')
T = strjust(S,'center')
```

Description `T = strjust(S)` or `T = strjust(S,'right')` returns a right-justified version of the character array S.

`T = strjust(S,'left')` returns a left-justified version of `S`.

`T = strjust(S,'center')` returns a center-justified version of `S`.

See Also `deblank`

strmatch

Purpose Find possible matches for a string

Syntax

```
x = strmatch('str',STRS)
x = strmatch('str',STRS,'exact')
```

Description `x = strmatch('`*`str`*`',STRS)` looks through the rows of the character array or cell array of strings `STRS` to find strings that begin with string *`str`*, returning the matching row indices. `strmatch` is fastest when `STRS` is a character array.

`x = strmatch('`*`str`*`',STRS,'exact')` returns only the indices of the strings in `STRS` matching *`str`* exactly.

Examples The statement

```
x = strmatch('max',strvcat('max','minimax','maximum'))
```

returns `x = [1; 3]` since rows 1 and 3 begin with `'max'`. The statement

```
x = strmatch('max',strvcat('max','minimax','maximum'),'exact')
```

returns `x = 1`, since only row 1 matches `'max'` exactly.

See Also `strcmp`, `strcmpi`, `strncmp`, `strncmpi`, `findstr`, `strvcat`

Purpose Compare the first `n` characters of two strings

Syntax

```
k = strncmp('str1','str2',n)
TF = strncmp(S,T,n)
```

Description `k = strncmp('`*`str1`*`','`*`str2`*`',n)` returns logical true (`1`) if the first `n` characters of the strings *`str1`* and *`str2`* are the same, and returns logical false (`0`) otherwise. Arguments *`str1`* and *`str2`* may also be cell arrays of strings.

`TF = strncmp(S,T,N)` where either `S` or `T` is a cell array of strings, returns an array `TF` the same size as `S` and `T` containing `1` for those elements of `S` and `T` that match (up to `n` characters), and `0` otherwise. `S` and `T` must be the same size (or one can be a scalar cell). Either one can also be a character array with the right number of rows.

Remarks The command `strncmp` is case sensitive. Any leading and trailing blanks in either of the strings are explicitly included in the comparison.

See Also `strcmp`, `strcmpi`, `strncmpi`, `strmatch`, `findstr`

strncmpi

Purpose

Compare first n characters of strings ignoring case

Syntax

```
strncmpi('str1','str2',n)
TF = strncmpi(S,T,n)
```

Description

`strncmpi('`*`str1`*`','`*`str2`*`',n)` returns 1 if the first `n` characters of the strings *`str1`* and *`str2`* are the same except for case, and 0 otherwise.

`TF = strncmpi(S,T,n)` when either `S` or `T` is a cell array of strings, returns an array the same size as `S` and `T` containing 1 for those elements of `S` and `T` that match except for case (up to `n` characters), and 0 otherwise. `S` and `T` must be the same size (or one can be a scalar cell). Either one can also be a character array with the right number of rows.

`strncmpi` supports international character sets.

See Also

`strncmp`, `strcmp`, `strcmpi`, `strmatch`, `findstr`

Purpose

Read formatted data from a string

Syntax

```
A = strread('str')
A = strread('str','',N)
A = strread('str','',param,value,...)
A = strread('str','',N,param,value,...)
[A,B,C,...] = strread('str','format')
[A,B,C,...] = strread('str','format',N)
[A,B,C,...] = strread('str','format',param,value,...)
[A,B,C,...] = strread('str','format',N,param,value,...)
```

Description

The first four syntaxes are used on strings containing only numeric data. If the input string, `str`, contains any text data, an error is generated.

`A = strread('str')` reads numeric data from the string, `str`, into the single variable `A`.

`A = strread('str','',N)` reads `N` lines of numeric data, where `N` is an integer greater than zero. If `N` is -1, `strread` reads the entire string.

`A = strread('str','',param,value,...)` customizes `strread` using `param`/`value` pairs, as listed in the table below.

`A = strread('str','',N,param,value,...)` reads `N` lines and customizes the `strread` using `param`/`value` pairs.

The next four syntaxes can be used on numeric or nonnumeric data. In this case, `strread` reads data from the string, `str`, into the variables `A`, `B`, `C`, and so on, using the specified `format`.

The type of each return argument is given by the `format` string. The number of return arguments must match the number of conversion specifiers in the `format` string. If there are fewer fields in the string than matching conversion specifiers in the `format` string, an error is generated.

The `format` string determines the number and types of return arguments. The number of return arguments is the number of items in the `format` string. The `format` string supports a subset of the conversion specifiers and conventions of the C language `fscanf` routine. Values for the `format` string are listed in the table below. Whitespace characters in the `format` string are ignored.

`[A,B,C,...] = strread('str','format')` reads data from the string, `str`, into the variables `A`, `B`, `C`, and so on, using the specified `format`, until the entire string is read.

format	Action	Output
Literals (ordinary characters)	Ignore the matching characters. For example, in a file that has `Dept` followed by a number (for department number), to skip the `Dept` and read only the number, use `'Dept'` in the `format` string.	None
`%d`	Read a signed integer value.	Double array
`%u`	Read an integer value.	Double array
`%f`	Read a floating point value.	Double array
`%s`	Read a whitespace-separated string.	Cell array of strings
`%q`	Read a string, which could be in double quotes.	Cell array of strings. Does not include the double quotes.
`%c`	Read characters, including white space.	Character array
`%[...]`	Read the longest string containing characters specified in the brackets.	Cell array of strings
`%[^...]`	Read the longest non-empty string containing characters that are not specified in the brackets.	Cell array of strings

format	Action	Output
`%*...` instead of `%`	Ignore the matching characters specified by `*`.	No output
`%w...` instead of `%`	Read field width specified by `w`. The `%f` format supports `%w.pf`, where `w` is the field width and `p` is the precision.	

`[A,B,C,...] = strread('str','format',N)` reads the data, reusing the format string `N` times, where `N` is an integer greater than zero. If `N` is -1, `strread` reads the entire string.

`[A,B,C,...] = strread('str','format',param,value,...)` customizes `strread` using `param`/`value` pairs, as listed in the table below.

`[A,B,C,...] = strread('str','format',N,param,value,...)` reads the data, reusing the format string `N` times and customizes the `strread` using `param`/`value` pairs.

param	value	Action
`whitespace`	`\*` where `*` can be:	Treats vector of characters, `*`, as whitespace. Default is `\b\r\n\t`.
	`b`	Backspace
	`f`	Form feed
	`n`	New line
	`r`	Carriage return
	`t`	Horizontal tab
	`\\`	Backslash
	`\''` or `''`	Single quotation mark
	`%%`	Percent sign
`delimiter`	Delimiter character	Specifies delimiter character. Default is none.
`expchars`	Exponent characters	Default is `eEdD`.

param	value	Action
bufsize	positive integer	Specifies the maximum string length, in bytes. Default is 4095.
headerlines	positive integer	Ignores the specified number of lines at the beginning of the file.
commentstyle	matlab	Ignores characters after %
commentstyle	shell	Ignores characters after #.
commentstyle	c	Ignores characters between /* and */.
commentstyle	c++	Ignores characters after //.

Remarks

If your data uses a character other than a space as a delimiter, you must use the strread parameter 'delimiter' to specify the delimiter. For example, if the string, str, used a semicolon as a delimiter, you would use this command.

```
[names,types,x,y,answer] = strread(str,'%s %s %f ...
    %d %s','delimiter',';')
```

Examples

```
s = sprintf('a,1,2\nb,3,4\n');
[a,b,c] = strread(s,'%s%d%d','delimiter',',')

a =
    'a'
    'b'

b =
     1
     3

c =
     2
     4
```

See Also

textread, sscanf

Purpose String search and replace

Syntax

```
str = strrep(str1,str2,str3)
```

Description `str = strrep(`*`str1`*`,`*`str2`*`,`*`str3`*`)` replaces all occurrences of the string *`str2`* within string *`str1`* with the string `str3`.

`strrep(`*`str1`*`,`*`str2`*`,`*`str3`*`)`, when any of *`str1`*, *`str2`*, or *`str3`* is a cell array of strings, returns a cell array the same size as *`str1`*, *`str2`* and *`str3`* obtained by performing a `strrep` using corresponding elements of the inputs. The inputs must all be the same size (or any can be a scalar cell). Any one of the strings can also be a character array with the right number of rows.

Examples

```
s1 = 'This is a good example.';
str = strrep(s1,'good','great')
str =
This is a great example.

A =
    'MATLAB'              'SIMULINK'
    'Toolboxes'           'The MathWorks'

B =
    'Handle Graphics'     'Real Time Workshop'
    'Toolboxes'           'The MathWorks'

C =
    'Signal Processing'    'Image Processing'
    'MATLAB'               'SIMULINK'

strrep(A,B,C)
ans =
    'MATLAB'    'SIMULINK'
    'MATLAB'    'SIMULINK'
```

See Also `findstr`

strtok

Purpose First token in string

Syntax

```
token = strtok('str',delimiter)
token = strtok('str')
[token,rem] = strtok(...)
```

Description

`token = strtok('str',delimiter)` returns the first token in the text string *`str`*, that is, the first set of characters before a delimiter is encountered. The vector `delimiter` contains valid delimiter characters. Any leading delimiters are ignored.

`token = strtok('str')` uses the default delimiters, the white space characters. These include tabs (ASCII 9), carriage returns (ASCII 13), and spaces (ASCII 32). Any leading white space characters are ignored.

`[token,rem] = strtok(...)` returns the remainder `rem` of the original string. The remainder consists of all characters from the first delimiter on.

Examples

```
s = '  This is a good example.';
[token,rem] = strtok(s)
token =
This
rem =
 is a good example.
```

See Also `findstr`, `strmatch`

Purpose

Create structure array

Syntax

```
s = struct('field1',{},'field2',{},...)
s = struct('field1',values1,'field2',values2,...)
```

Description

`s = struct('field1',{},'field2',{},...)` creates an empty structure with fields `field1`, `field2`, ...

`s = struct('field1',values1,'field2',values2,...)` creates a structure array with the specified fields and values. The value arrays `values1`, `values2`, etc. must be cell arrays of the same size or scalar cells. Corresponding elements of the value arrays are placed into corresponding structure array elements. The size of the resulting structure is the same size as the value cell arrays or 1-by-1 if none of the values is a cell.

Examples

The command

```
s = struct('type',{'big','little'},'color',{'red'},'x',{3 4})
```

produces a structure array `s`:

```
s =
1x2 struct array with fields:
    type
    color
    x
```

The value arrays have been distributed among the fields of `s`:

```
s(1)
ans =
     type: 'big'
    color: 'red'
        x: 3

s(2)
ans =
     type: 'little'
    color: 'red'
        x: 4
```

Similarly, the command

```
a.b = struct('z',{});
```

produces an empty structure `a.b` with field `z`.

```
a.b
ans =
      0x0 struct array with fields:
          z
```

See Also `fieldnames`, `getfield`, `rmfield`, `setfield`

Purpose Structure to cell array conversion

Syntax

```
c = struct2cell(s)
```

Description `c = struct2cell(s)` converts the `m`-by-`n` structure `s` (with `p` fields) into a `p`-by-`m`-by-`n` cell array `c`.

If structure `s` is multidimensional, cell array `c` has size `[p size(s)]`.

Examples The commands

```
clear s, s.category = 'tree';
s.height = 37.4; s.name = 'birch';
```

create the structure

```
s =
    category: 'tree'
      height: 37.4000
        name: 'birch'
```

Converting the structure to a cell array,

```
c = struct2cell(s)

c =
    'tree'
    [37.4000]
    'birch'
```

See Also `cell2struct`, `fieldnames`

strvcat

Purpose Vertical concatenation of strings

Syntax

```
S = strvcat(t1,t2,t3,...)
```

Description `S = strvcat(t1,t2,t3,...)` forms the character array `S` containing the text strings (or string matrices) `t1,t2,t3,...` as rows. Spaces are appended to each string as necessary to form a valid matrix. Empty arguments are ignored.

Remarks If each text parameter, `ti`, is itself a character array, `strvcat` appends them vertically to create arbitrarily large string matrices.

Examples The command `strvcat('Hello','Yes')` is the same as `['Hello';'Yes  ']`, except that `strvcat` performs the padding automatically.

```
t1 = 'first';t2 = 'string';t3 = 'matrix';t4 = 'second';

S1 = strvcat(t1,t2,t3)          S2 = strvcat(t4,t2,t3)

S1 =                            S2 =

first                           second
string                          string
matrix                          matrix

S3 = strvcat(S1,S2)

S3 =
first
string
matrix
second
string
matrix
```

See Also `cat`, `int2str`, `mat2str`, `num2str`, `strings`

Purpose

Single index from subscripts

Syntax

```
IND = sub2ind(siz,I,J)
IND = sub2ind(siz,I1,I2,...,In)
```

Description

The `sub2ind` command determines the equivalent single index corresponding to a set of subscript values.

`IND = sub2ind(siz,I,J)` returns the linear index equivalent to the row and column subscripts `I` and `J` for a matrix of size `siz`.

`IND = sub2ind(siz,I1,I2,...,In)` returns the linear index equivalent to the `n` subscripts `I1`,`I2`,...,`In` for an array of size `siz`.

Examples

Create a 3-by-4-by-2 matrix, `A`.

```
A = [17 24 1 8; 2 22 7 14; 4 6 13 20];
A(:,:,2) = A - 10

A(:,:,1) =

    17    24     1     8
     2    22     7    14
     4     6    13    20


A(:,:,2) =

     7    14    -9    -2
    -8    12    -3     4
    -6    -4     3    10
```

The value at row 2, column 1, page 2 of the matrix is -8.

```
A(2,1,2)

ans =

    -8
```

To convert `A(2,1,2)` into its equivalent single subscript, use `sub2ind`.

```
sub2ind(size(A),2,1,2)

ans =

    14
```

You can now access the same location in A using the single subscripting method.

```
A(14)

ans =

    -8
```

See Also ind2sub, find

Purpose Create and control multiple axes

Syntax

```
subplot(m,n,p)
subplot(m,n,p,'replace')
subplot(h)
subplot('Position',[left bottom width height])
h = subplot(...)
```

Description

`subplot` divides the current figure into rectangular panes that are numbered row-wise. Each pane contains an axes. Subsequent plots are output to the current pane.

`subplot(m,n,p)` creates an axes in the `p`-th pane of a figure divided into an `m`-by-`n` matrix of rectangular panes. The new axes becomes the current axes. If `p` is a vector, specifies an axes having a position that covers all the subplot positions listed in `p`.

`subplot(m,n,p,'replace')` If the specified axes already exists, delete it and creat a new axes.

`subplot(h)` makes the axes with handle `h` current for subsequent plotting commands.

`subplot('Position',[left bottom width height])` creates an axes at the position specified by a four-element vector. `left`, `bottom`, `width`, and `height` are in normalized coordinates in the range from 0.0 to 1.0.

`h = subplot(...)` returns the handle to the new axes.

Remarks

If a `subplot` specification causes a new axes to overlap any existing axes, then `subplot` deletes the existing axes. However, if the `subplot` specification exactly matches the position of an existing axes, then the matching axes is not deleted and it becomes the current axes.

`subplot(1,1,1)` or `clf` deletes all axes objects and returns to the default `subplot(1,1,1)` configuration.

You can omit the parentheses and specify subplot as.

```
subplot mnp
```

where m refers to the row, n refers to the column, and p specifies the pane.

Special Case – subplot(111)

The command subplot(111) is not identical in behavior to subplot(1,1,1) and exists only for compatibility with previous releases. This syntax does not immediately create an axes, but instead sets up the figure so that the next graphics command executes a clf reset (deleting all figure children) and creates a new axes in the default position. This syntax does not return a handle, so it is an error to specify a return argument. (This behavior is implemented by setting the figure's NextPlot property to replace.)

Examples

To plot income in the top half of a figure and outgo in the bottom half,

```
income = [3.2 4.1 5.0 5.6];
outgo = [2.5 4.0 3.35 4.9];
subplot(2,1,1); plot(income)
subplot(2,1,2); plot(outgo)
```

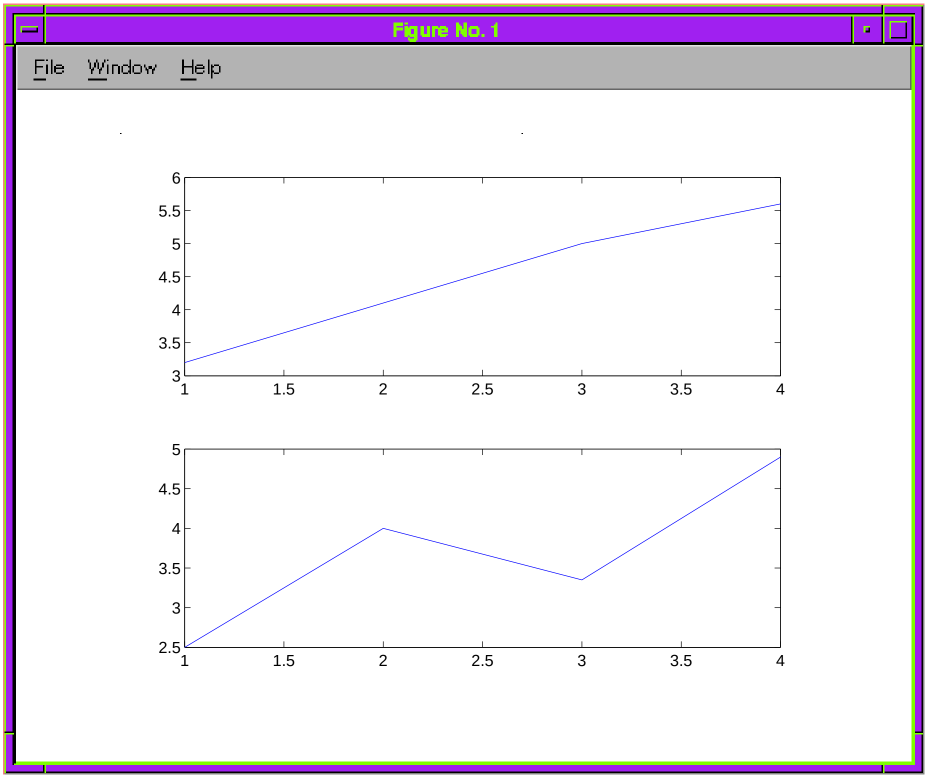
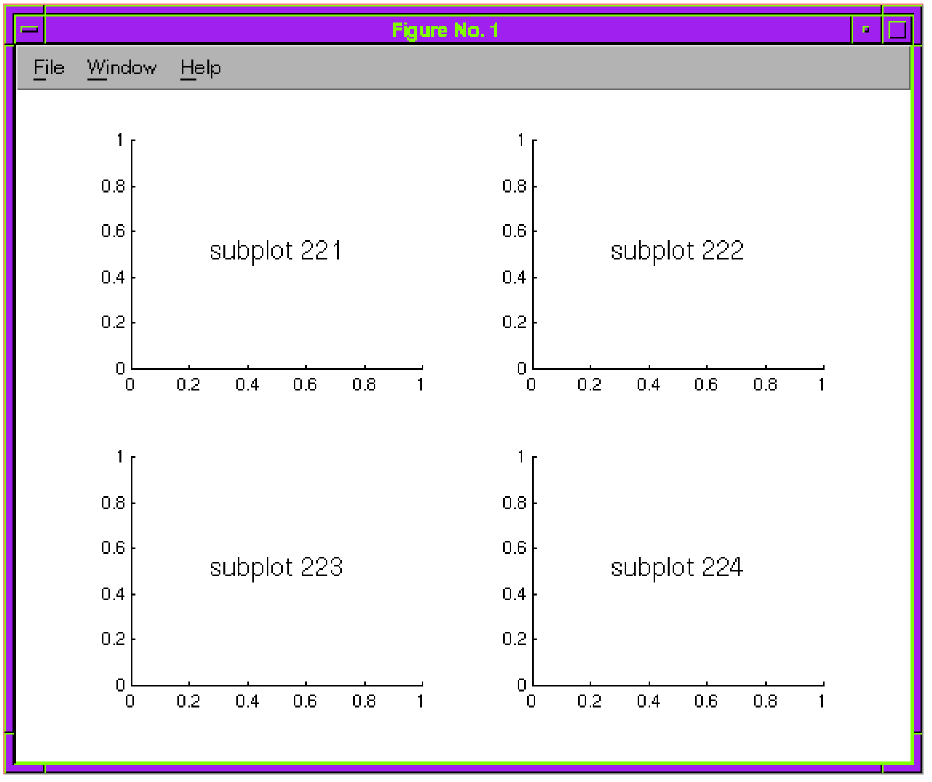


Figure No. 1
File
Window
Help
6
5.5
5
4.5
4
3.5
3
1
1.5
2
2.5
3
3.5
4
5
4.5
4
3.5
3
2.5
1
1.5
2
2.5
3
3.5
4

The following illustration shows four subplot regions and indicates the command used to create each.



See Also

`axes`, `cla`, `clf`, `figure`, `gca`

subsasgn

Purpose

Overloaded method for `A(I)=B`, `A{I}=B`, and `A.field=B`

Syntax

```
A = subsasgn(A,S,B)
```

Description

`A = subsasgn(A,S,B)` is called for the syntax `A(i)=B`, `A{i}=B`, or `A.i=B` when `A` is an object. `S` is a structure array with the fields:

- `type`: A string containing `'()'`, `'{}'`, or `'.'`, where `'()'` specifies integer subscripts; `'{}'` specifies cell array subscripts, and `'.'` specifies subscripted structure fields.
- `subs`: A cell array or string containing the actual subscripts.

Remarks

`subsasgn` is designed to be used by the MATLAB interpreter to handle indexed assignments to objects. Calling `subsasgn` directly as a function is not recommended. If you do use `subsasgn` in this way, it conforms to the formal MATLAB dispatching rules and may yield unexpected results.

Examples

The syntax `A(1:2,:)=B` calls `A=subsasgn(A,S,B)` where `S` is a 1-by-1 structure with `S.type='()'` and `S.subs = {1:2,':'}`. A colon used as a subscript is passed as the string `':'`.

The syntax `A{1:2}=B` calls `A=subsasgn(A,S,B)` where `S.type='{}'`.

The syntax `A.field=B` calls `subsasgn(A,S,B)` where `S.type='.'` and `S.subs='field'`.

These simple calls are combined in a straightforward way for more complicated subscripting expressions. In such cases `length(S)` is the number of subscripting levels. For instance, `A(1,2).name(3:5)=B` calls `A=subsasgn(A,S,B)` where `S` is 3-by-1 structure array with the following values:

`S(1).type='()'`	`S(2).type='.'`	`S(3).type='()'`
`S(1).subs={1,2}`	`S(2).subs='name'`	`S(3).subs={3:5}`

See Also

`subsref`

See “Handling Subscripted Assignment” for more information about overloaded methods and `subsasgn`.

subsindex

Purpose Overloaded method for `X(A)`

Syntax

```
ind = subsindex(A)
```

Description `ind = subsindex(A)` is called for the syntax `'X(A)'` when `A` is an object. `subsindex` must return the value of the object as a zero-based integer index. (`ind` must contain integer values in the range `0` to `prod(size(X))-1`). `subsindex` is called by the default `subsref` and `subsasgn` functions, and you can call it if you overload these functions.

See Also `subsasgn`, `subsref`

Purpose

Angle between two subspaces

Syntax

```
theta = subspace(A,B)
```

Description

`theta = subspace(A,B)` finds the angle between two subspaces specified by the columns of `A` and `B`. If `A` and `B` are column vectors of unit length, this is the same as `acos(A'*B)`.

Remarks

If the angle between the two subspaces is small, the two spaces are nearly linearly dependent. In a physical experiment described by some observations `A`, and a second realization of the experiment described by `B`, `subspace(A,B)` gives a measure of the amount of new information afforded by the second experiment not associated with statistical errors of fluctuations.

Examples

Consider two subspaces of a Hadamard matrix, whose columns are orthogonal.

```
H = hadamard(8);
A = H(:,2:4);
B = H(:,5:8);
```

Note that matrices `A` and `B` are different sizes— `A` has three columns and `B` four. It is not necessary that two subspaces be the same size in order to find the angle between them. Geometrically, this is the angle between two hyperplanes embedded in a higher dimensional space.

```
theta = subspace(A,B)
theta =
    1.5708
```

That `A` and `B` are orthogonal is shown by the fact that `theta` is equal to $\pi/2$.

```
theta - pi/2
ans =
     0
```

subsref

Purpose

Overloaded method for `A(I)`, `A{I}` and `A.field`

Syntax

```
B = subsref(A,S)
```

Description

`B = subsref(A,S)` is called for the syntax `A(i)`, `A{i}`, or `A.i` when `A` is an object. `S` is a structure array with the fields:

- `type`: A string containing `'()'`, `'{}'`, or `'.'`, where `'()'` specifies integer subscripts; `'{}'` specifies cell array subscripts, and `'.'` specifies subscripted structure fields.
- `subs`: A cell array or string containing the actual subscripts.

Remarks

`subsref` is designed to be used by the MATLAB interpreter to handle indexed references to objects. Calling `subsref` directly as a function is not recommended. If you do use `subsref` in this way, it conforms to the formal MATLAB dispatching rules and may yield unexpected results.

Examples

The syntax `A(1:2,:)` calls `subsref(A,S)` where `S` is a 1-by-1 structure with `S.type='()'` and `S.subs={1:2,':'}`. A colon used as a subscript is passed as the string `':'`.

The syntax `A{1:2}` calls `subsref(A,S)` where `S.type='{}'` and `S.subs={1:2}`.

The syntax `A.field` calls `subsref(A,S)` where `S.type='.'` and `S.subs='field'`.

These simple calls are combined in a straightforward way for more complicated subscripting expressions. In such cases `length(S)` is the number of subscripting levels. For instance, `A(1,2).name(3:5)` calls `subsref(A,S)` where `S` is 3-by-1 structure array with the following values:

```
S(1).type='()'       S(2).type='.'        S(3).type='()'
S(1).subs={1,2}      S(2).subs='name'     S(3).subs={3:5}
```

See Also

`subsasgn`

See “Handling Subscripted Reference” for more information about overloaded methods and `subsref`.

Purpose

Create structure argument for subsasgn or subsref

Syntax

```
S = substruct(type1,subs1,type2,subs2,...)
```

Description

S = substruct(type1,subs1,type2,subs2,...) creates a structure with the fields required by an overloaded subsref or subsasgn method. Each type string must be one of '.', '()', or '{}'. The corresponding subs argument must be either a field name (for the '.' type) or a cell array containing the index vectors (for the '()' or '{}' types).

The output S is a structure array containing the fields:

- type – one of '.', '()', or '{}'
- subs – subscript values (field name or cell array of index vectors)

Examples

To call subsref with parameters equivalent to the syntax

```
B = A(3,5).field
```

you can use

```
S = substruct('()',{3,5},'.','field');
B = subsref(A,S);
```

The structure created by substruct in this example contains the following.

```
S(1)

ans =

    type: '()'
    subs: {[3]  [5]}

S(2)

ans =

    type: '.'
    subs: 'field'
```

See Also

subsasgn, subsref

subvolume

Purpose Extract subset of volume data set

Syntax

```
[Nx,Ny,Nz,Nv] = subvolume(X,Y,Z,V,limits)
[Nx,Ny,Nz,Nv] = subvolume(V,limits)
Nv = subvolume(...)
```

Description

`[Nx,Ny,Nz,Nv] = subvolume(X,Y,Z,V,limits)` extracts a subset of the volume data set `V` using the specified axis-aligned `limits`. `limits = [xmin,xmax,ymin, ymax,zmin,zmax]` (Any `NaN`s in the limits indicate that the volume should not be cropped along that axis).

The arrays `X`, `Y`, and `Z` define the coordinates for the volume `V`. The subvolume is returned in `NV` and the coordinates of the subvolume are given in `NX`, `NY`, and `NZ`.

`[Nx,Ny,Nz,Nv] = subvolume(V,limits)` assumes the arrays `X`, `Y`, and `Z` are defined as `[X,Y,Z] = meshgrid(1:N,1:M,1:P)` where `[M,N,P] = size(V)`.

`Nv = subvolume(...)` returns only the subvolume.

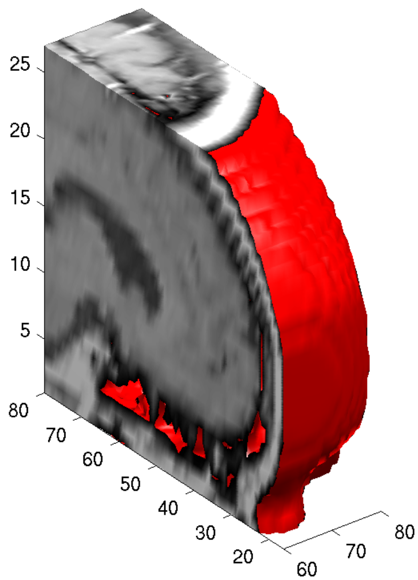
Examples

This example uses a data set that is a collection of MRI slices of a human skull. The data is processed in a variety of ways:

- The 4-D array is squeezed (`squeeze`) into three dimensions and then a subset of the data is extracted (`subvolume`).
- The outline of the skull is an isosurface generated as a patch (`p1`) whose vertex normals are recalculated to improve the appearance when lighting is applied (`patch`, `isosurface`, `isonormals`).
- A second patch (`p2`) with interpolated face color draws the end caps (`FaceColor`, `isocaps`).
- The view of the object is set (`view`, `axis`, `daspect`).
- A 100-element grayscale colormap provides coloring for the end caps (`colormap`).
- Adding lights to the right and left of the camera illuminates the object (`camlight`, `lighting`).

```
load mri
D = squeeze(D);
[x,y,z,D] = subvolume(D,[60,80,nan,80,nan,nan]);
```

```
p1 = patch(isosurface(x,y,z,D, 5),...
     'FaceColor','red','EdgeColor','none');
isonormals(x,y,z,D,p1);
p2 = patch(isocaps(x,y,z,D, 5),...
     'FaceColor','interp','EdgeColor','none');
view(3); axis tight; daspect([1,1,.4])
colormap(gray(100))
camlight right; camlight left; lighting gouraud
```



See Also

`isocaps`, `isonormals`, `isosurface`, `reducepatch`, `reducevolume`, `smooth3`

sum

Purpose Sum of array elements

Syntax

```
B = sum(A)
B = sum(A,dim)
```

Description `B = sum(A)` returns sums along different dimensions of an array.

If `A` is a vector, `sum(A)` returns the sum of the elements.

If `A` is a matrix, `sum(A)` treats the columns of `A` as vectors, returning a row vector of the sums of each column.

If `A` is a multidimensional array, `sum(A)` treats the values along the first non-singleton dimension as vectors, returning an array of row vectors.

`B = sum(A,dim)` sums along the dimension of `A` specified by scalar `dim`.

Remarks `sum(diag(X))` is the `trace` of `X`.

Examples The magic square of order 3 is

```
M = magic(3)
M =
     8    1    6
     3    5    7
     4    9    2
```

This is called a magic square because the sums of the elements in each column are the same.

```
sum(M) =
     15    15    15
```

as are the sums of the elements in each row, obtained by transposing:

```
sum(M') =
     15    15    15
```

See Also `cumsum`, `diff`, `prod`, `trace`

Purpose Superior class relationship

Syntax

```
superiorto('class1','class2',...)
```

Description The `superiorto` function establishes a hierarchy that determines the order in which MATLAB calls object methods.

`superiorto('class1','class2',...)` invoked within a class constructor method (say `myclass.m`) indicates that `myclass`'s method should be invoked if a function is called with an object of class `myclass` and one or more objects of class `class1`, `class2`, and so on.

Remarks Suppose `A` is of class `'class_a'`, `B` is of class `'class_b'` and `C` is of class `'class_c'`. Also suppose the constructor `class_c.m` contains the statement: `superiorto('class_a')`. Then `e = fun(a,c)` or `e = fun(c,a)` invokes `class_c/fun`.

If a function is called with two objects having an unspecified relationship, the two objects are considered to have equal precedence, and the leftmost object's method is called. So, `fun(b,c)` calls `class_b/fun`, while `fun(c,b)` calls `class_c/fun`.

See Also `inferiorto`

support

Purpose Open MathWorks Technical Support Web page

Syntax

```
support
```

Description `support` opens your web browser to The MathWorks Technical Support Web page at `http://www.mathworks.com/support`.

This page contains the following items:

- A Solution Search Engine
- The "Virtual Technical Support Engineer" that, through a series of questions, determines possible solutions to the problems you are experiencing
- Technical Notes
- Tutorials
- Bug fixes and patches

See Also `web`

Purpose

3-D shaded surface plot

Syntax

```
surf(Z)
surf(X,Y,Z)
surf(X,Y,Z,C)
surf(...,'PropertyName',PropertyValue)
surfc(...)
h = surf(...)
h = surfc(...)
```

Description

Use `surf` and `surfc` to view mathematical functions over a rectangular region. `surf` and `surfc` create colored parametric surfaces specified by `X`, `Y`, and `Z`, with color specified by `Z` or `C`.

`surf(Z)` creates a a three-dimensional shaded surface from the *z* components in matrix `Z`, using `x = 1:n` and `y = 1:m`, where `[m,n] = size(Z)`. The height, `Z`, is a single-valued function defined over a geometrically rectangular grid. `Z` specifies the color data as well as surface height, so color is proportional to surface height.

`surf(X,Y,Z)` creates a shaded surface using `Z` for the color data as well as surface height. `X` and `Y` are vectors or matrices defining the *x* and *y* components of a surface. If `X` and `Y` are vectors, `length(X) = n` and `length(Y) = m`, where `[m,n] = size(Z)`. In this case, the vertices of the surface faces are $(X(j), Y(i), Z(i, j))$ triples.

`surf(X,Y,Z,C)` creates a shaded surface, with color defined by `C`. MATLAB performs a linear transformation on this data to obtain colors from the current colormap.

`surf(...,'PropertyName',PropertyValue)` specifies surface properties along with the data.

`surfc(...)` draws a contour plot beneath the surface.

`h = surf(...)` and `h = surfc(...)` return a handle to a surface graphics object.

Algorithm

Abstractly, a parametric surface is parametrized by two independent variables, i and j, which vary continuously over a rectangle; for example, $1 \le i \le m$ and $1 \le j \le n$. The three functions, x(i,j), y(i,j), and z(i,j), specify the surface. When i and j are integer values, they define a rectangular grid with integer grid points. The functions x(i,j), y(i,j), and z(i,j) become three m-by-n matrices, X, Y and Z. surface color is a fourth function, c(i,j), denoted by matrix C.

Each point in the rectangular grid can be thought of as connected to its four nearest neighbors.

```
         i–1,j
           |
  i,j–1 – i,j – i,j+1
           |
         i+1,j
```

This underlying rectangular grid induces four-sided patches on the surface. To express this another way, [X(:) Y(:) Z(:)] returns a list of triples specifying points in 3-space. Each interior point is connected to the four neighbors inherited from the matrix indexing. Points on the edge of the surface have three neighbors; the four points at the corners of the grid have only two neighbors. This defines a mesh of quadrilaterals or a *quad-mesh*.

Surface color can be specified in two different ways – at the vertices or at the centers of each patch. In this general setting, the surface need not be a single-valued function of x and y. Moreover, the four-sided surface patches need not be planar. For example, you can have surfaces defined in polar, cylindrical, and spherical coordinate systems.

The shading function sets the shading. If the shading is interp, C must be the same size as X, Y, and Z; it specifies the colors at the vertices. The color within a surface patch is a bilinear function of the local coordinates. If the shading is faceted (the default) or flat, C(i,j) specifies the constant color in the surface patch:

```
   (i,j)   –   (i,j+1)
     |   C(i,j)  |
  (i+1,j)  –  (i+1,j+1)
```

In this case, C can be the same size as X, Y, and Z and its last row and column are ignored, Alternatively, its row and column dimensions can be one less than those of X, Y, and Z.

The surf and surfc functions specify the view point using view(3).

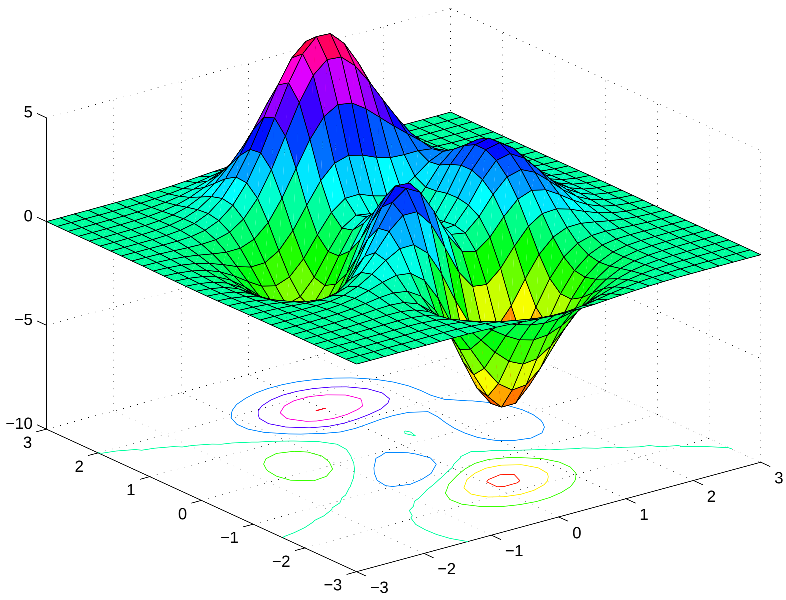
The range of X, Y, and Z, or the current setting of the axes XLimMode, YLimMode, and ZLimMode properties (also set by the axis function) determine the axis labels.

The range of C, or the current setting of the axes CLim and ClimMode properties (also set by the caxis function) determine the color scaling. The scaled color values are used as indices into the current colormap.

Examples

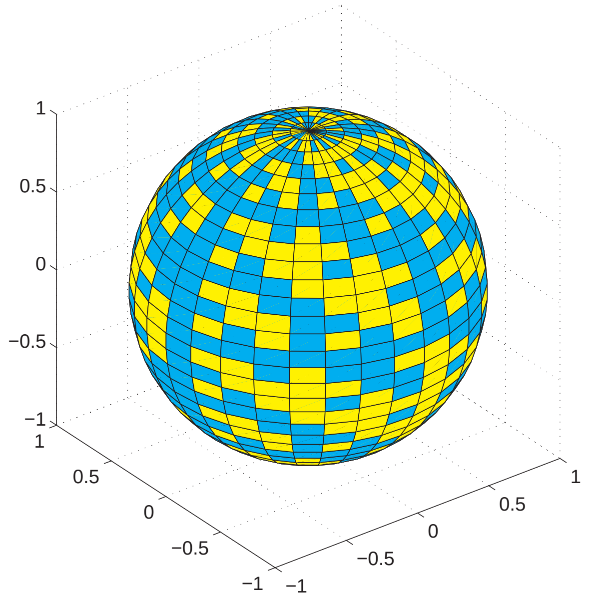
Display a surface and contour plot of the peaks surface.

```
[X,Y,Z] = peaks(30);
surfc(X,Y,Z)
colormap hsv
axis([–3 3 –3 3 –10 5])
```



Color a sphere with the pattern of +1s and -1s in a Hadamard matrix.

```
k = 5;
n = 2^k-1;
[x,y,z] = sphere(n);
c = hadamard(2^k);
surf(x,y,z,c);
colormap([1  1  0; 0  1  1])
axis equal
```



See Also

axis, caxis, colormap, contour, mesh, pcolor, shading, view

Properties for surface graphics objects

Purpose

Convert surface data to patch data

Syntax

```
fvc = surf2patch(h)
fvc = surf2patch(Z)
fvc = surf2patch(Z,C)
fvc = surf2patch(X,Y,Z)
fvc = surf2patch(X,Y,Z,C)
fvc = surf2patch(...,'triangles')
[f,v,c] = surf2patch(...)
```

Description

`fvc = surf2patch(h)` converts the geometry and color data from the `surface` object identified by the handle `h` into patch format and returns the face, vertex, and color data in the struct `fvc`. You can pass this struct directly to the `patch` command.

`fvc = surf2patch(Z)` calculates the patch data from the surface's `ZData` matrix `Z`.

`fvc = surf2patch(Z,C)` calculates the patch data from the surface's `ZData` and `CData` matrices `Z` and `C`.

`fvc = surf2patch(X,Y,Z)` calculates the patch data from the surface's `XData`, `YData`, and `ZData` matrices `X`, `Y`, and `Z`.

`fvc = surf2patch(X,Y,Z,C)` calculates the patch data from the surface's `XData`, `YData`, `ZData`, and `CData` matrices `X`, `Y`, `Z`, and `C`.

`fvc = surf2patch(...,'triangles')` creates triangular faces instead of the quadrilaterals that compose surfaces.

`[f,v,c] = surf2patch(...)` returns the face, vertex, and color data in the three arrays `f`, `v`, and `c` instead of a struct.

Examples

The first example uses the `sphere` command to generate the `XData`, `YData`, and `ZData` of a surface, which is then converted to a patch. Note that the `ZData` (`z`) is passed to `surf2patch` as both the third and fourth arguments – the third argument is the `ZData` and the fourth argument is taken as the `CData`. This is because the `patch` command does not automatically use the z-coordinate data for the color data, as does the `surface` command.

Also, because patch is a low-level command, you must set the view to 3-D and shading to faceted to produce the same results produced by the surf command.

```
[x y z] = sphere;
patch(surf2patch(x,y,z,z));
shading faceted; view(3)
```

In the second example surf2patch calculates face, vertex, and color data from a surface whose handle has been passed as an argument.

```
s = surf(peaks);
pause
patch(surf2patch(s));
delete(s)
shading faceted; view(3)
```

See Also

patch, reducepatch, shrinkfaces, surface, surf

Purpose Create surface object

Syntax

```
surface(Z)
surface(Z,C)
surface(X,Y,Z)
surface(X,Y,Z,C)
surface(...'PropertyName',PropertyValue,...)
h = surface(...)
```

Description

`surface` is the low-level function for creating surface graphics objects. surfaces are plots of matrix data created using the row and column indices of each element as the *x*- and *y*-coordinates and the value of each element as the z-coordinate.

`surface(Z)` plots the surface specified by the matrix `Z`. Here, `Z` is a single-valued function, defined over a geometrically rectangular grid.

`surface(Z,C)` plots the surface specified by `Z` and colors it according to the data in `C` (see "Examples").

`surface(X,Y,Z)` uses `C = Z`, so color is proportional to surface height above the *x*-*y* plane.

`surface(X,Y,Z,C)` plots the parametric surface specified by `X`, `Y` and `Z`, with color specified by `C`.

`surface(x,y,Z)`, `surface(x,y,Z,C)` replaces the first two matrix arguments with vectors and must have `length(x) = n` and `length(y) = m` where `[m,n] = size(Z)`. In this case, the vertices of the surface facets are the triples `(x(j),y(i),Z(i,j))`. Note that `x` corresponds to the columns of `Z` and `y` corresponds to the rows of `Z`. For a complete discussion of parametric surfaces, see the `surf` function.

`surface(...'`*`PropertyName`*`',PropertyValue,...)` follows the `X`, `Y`, `Z`, and `C` arguments with property name/property value pairs to specify additional surface properties. These properties are described in the "Surface Properties" section.

`h = surface(...)` returns a handle to the created surface object.

Remarks

Unlike high-level area creation functions, such as `surf` or `mesh`, `surface` does not respect the settings of the figure and axes `NextPlot` properties. It simply adds the surface object to the current axes.

If you do not specify separate color data (`C`), MATLAB uses the matrix (`Z`) to determine the coloring of the surface. In this case, color is proportional to values of `Z`. You can specify a separate matrix to color the surface independently of the data defining the area of the surface.

You can specify properties as property name/property value pairs, structure arrays, and cell arrays (see `set` and `get` for examples of how to specify these data types).

`surface` provides convenience forms that allow you to omit the property name for the `XData`, `YData`, `ZData`, and `CData` properties. For example,

```
surface('XData',X,'YData',Y,'ZData',Z,'CData',C)
```

is equivalent to:

```
surface(X,Y,Z,C)
```

When you specify only a single matrix input argument,

```
surface(Z)
```

MATLAB assigns the data properties as if you specified,

```
surface('XData',[1:size(Z,2)],...
        'YData',[1:size(Z,1)],...
        'ZData',Z,...
        'CData',Z)
```

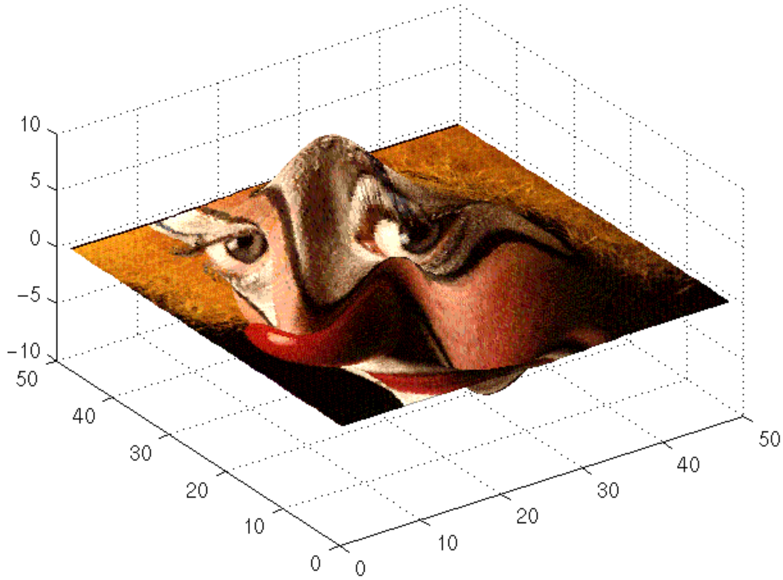
The `axis`, `caxis`, `colormap`, `hold`, `shading`, and `view` commands set graphics properties that affect surfaces. You can also set and query surface property values after creating them using the `set` and `get` commands.

Example

This example creates a surface using the `peaks` M-file to generate the data, and colors it using the clown image. The `ZData` is a 49-by-49 element matrix, while the `CData` is a 200-by-320 matrix. You must set the surface's `FaceColor` to `texturemap` to use `ZData` and `CData` of different dimensions.

```
load clown
surface(peaks,flipud(X),...
```

```
        'FaceColor','texturemap',...
        'EdgeColor','none',...
        'CDataMapping','direct')
colormap(map)
view(-35,45)
```



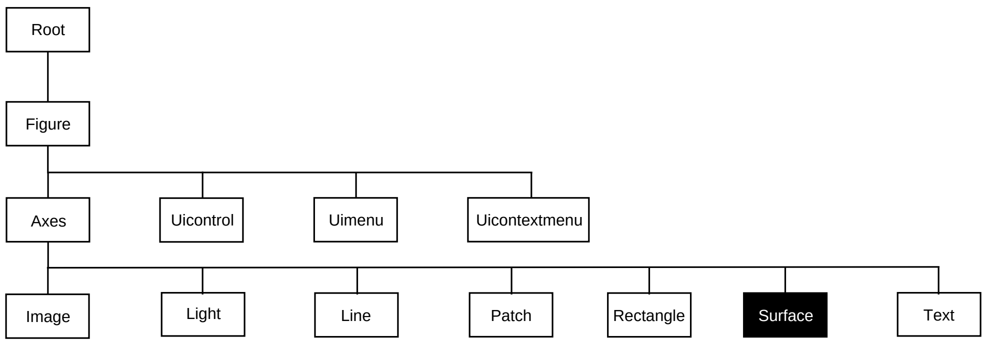
Note the use of the `surface(Z,C)` convenience form combined with property name/property value pairs.

Since the clown data (`X`) is typically viewed with the `image` command, which MATLAB normally displays with `'ij'` axis numbering and `direct` `CDataMapping`, this example reverses the data in the vertical direction using `flipud` and sets the `CDataMapping` property to `direct`.

See Also

`ColorSpec`, `mesh`, `patch`, `pcolor`, `surf`

Object Hierarchy



Setting Default Properties

You can set default surface properties on the axes, figure, and root levels.

```
set(0,'DefaultSurfaceProperty',PropertyValue...)
set(gcf,'DefaultSurfaceProperty',PropertyValue...)
set(gca,'DefaultSurfaceProperty',PropertyValue...)
```

Where *`Property`* is the name of the surface property whose default value you want to set and `PropertyValue` is the value you are specifying. Use `set` and `get` to access the surface properties.

Property List

The following table lists all surface properties and provides a brief description of each. The property name links take you to an expanded description of the properties.

Property Name	Property Description	Property Value
Data Defining the Object		
`XData`	The *x*-coordinates of the vertices of the surface	Values: vector or matrix
`YData`	The *y*-coordinates of the vertices of the surface	Values: vector or matrix

Property Name	Property Description	Property Value
ZData	The z-coordinates of the vertices of the surface	Values: matrix
Specifying Color		
CData	Color data	Values: scalar, vector, or matrix Default: [] empty matrix
CDataMapping	Controls mapping of CData to colormap	Values: scaled, direct Default: scaled
EdgeColor	Color of face edges	Values: ColorSpec, none, flat, interp Default: ColorSpec
FaceColor	Color of face	Values: ColorSpec, none, flat, interp Default: ColorSpec
MarkerEdgeColor	Color of marker or the edge color for filled markers	Values: ColorSpec, none, auto Default: auto
MarkerFaceColor	Fill color for markers that are closed shapes	Values: ColorSpec, none, auto Default: none
Specifying Transparency		
AlphaData	The transparency data	m-by-n matrix of double or uint8
AlphaDataMapping	Transparency mapping method	none, direct, scaled Default: scaled
EdgeAlpha	Transparency of the edges of patch faces	scalar, flat, interp Default: 1 (opaque)

Property Name	Property Description	Property Value
FaceAlpha	Transparency of the patch face	scalar, flat, interp, texture Default: 1 (opaque)
Controlling the Effects of Lights		
AmbientStrength	Intensity of the ambient light	Values: scalar >=0 and <=1 Default: 0.3
BackFaceLighting	Controls lighting of faces pointing away from camera	Values: unlit, lit, reverselit Default: reverselit
DiffuseStrength	Intensity of diffuse light	Values: scalar >=0 and <=1 Default: 0.6
EdgeLighting	Method used to light edges	Values: none, flat, gouraud, phong Default: none
FaceLighting	Method used to light edges	Values: none, flat, gouraud, phong Default: none
NormalMode	MATLAB-generated or user-specified normal vectors	Values: auto, manual Default: auto
SpecularColorReflectance	Composite color of specularly reflected light	Values: scalar 0 to 1 Default: 1
SpecularExponent	Harshness of specular reflection	Values: scalar >= 1 Default: 10
SpecularStrength	Intensity of specular light	Values: scalar >=0 and <=1 Default: 0.9
VertexNormals	Vertex normal vectors	Values: matrix

Property Name	Property Description	Property Value
Defining Edges and Markers		
LineStyle	Select from five line styles.	Values: –, ––, :, –., none Default: –
LineWidth	The width of the edge in points	Values: scalar Default: 0.5 points
Marker	Marker symbol to plot at data points	Values: see Marker property Default: none
MarkerSize	Size of marker in points	Values: size in points Default: 6
Controlling the Appearance		
Clipping	Clipping to axes rectangle	Values: on, off Default: on
EraseMode	Method of drawing and erasing the surface (useful for animation)	Values: normal, none, xor, background Default: normal
MeshStyle	Specifies whether to draw all edge lines or just row or column edge lines	Values: both, row, column Defaults: both
SelectionHighlight	Highlight surface when selected (Selected property set to on)	Values: on, off Default: on
Visible	Make the surface visible or invisible	Values: on, off Default: on
Controlling Access to Objects		
HandleVisibility	Determines if and when the the surface's handle is visible to other functions	Values: on, callback, off Default: on
HitTest	Determines if the surface can become the current object (see the figure CurrentObject property)	Values: on, off Default: on

Property Name	Property Description	Property Value
Properties Related to Callback Routine Execution		
`BusyAction`	Specifies how to handle callback routine interruption	Values: `cancel`, `queue` Default: `queue`
`ButtonDownFcn`	Defines a callback routine that executes when a mouse button is pressed on over the surface	Values: string Default: `''` (empty string)
`CreateFcn`	Defines a callback routine that executes when an surface is created	Values: string Default: `''` (empty string)
`DeleteFcn`	Defines a callback routine that executes when the surface is deleted (via `close` or `delete`)	Values: string Default: `''` (empty string)
`Interruptible`	Determines if callback routine can be interrupted	Values: `on`, `off` Default: `on` (can be interrupted)
`UIContextMenu`	Associates a context menu with the surface	Values: handle of a uicontextmenu
General Information About the Surface		
`Children`	Surface objects have no children	Values: `[]` (empty matrix)
`Parent`	The parent of a surface object is always an axes object	Value: axes handle
`Selected`	Indicates whether the surface is in a “selected” state.	Values: `on`, `off` Default: `on`
`Tag`	User-specified label	Value: any string Default: `''` (empty string)
`Type`	The type of graphics object (read only)	Value: the string `'surface'`
`UserData`	User-specified data	Values: any matrix Default: `[]` (empty matrix)

Modifying Properties

You can set and query graphics object properties in two ways:

- The Property Editor is an interactive tool that enables you to see and change object property values.
- The `set` and `get` commands enable you to set and query the values of properties

To change the default value of properties see Settingcreating_plots Default Property Values.

Surface Property Descriptions

This section lists property names along with the types of values each accepts. Curly braces { } enclose default values.

`AlphaData` m-by-n matrix of `double` or `uint8`

The transparency data. A matrix of non-NaN values specifying the transparency of each face or vertex of the object. The `AlphaData` can be of class double or uint8.

MATLAB determines the transparency in one of three ways:

- Using the elements of `AlphaData` as transparency values (`AlphaDataMapping` set to `none`).
- Using the elements of `AlphaData` as indices into the current alphamap (`AlphaDataMapping` set to `direct`).
- Scaling the elements of `AlphaData` to range between the minimum and maximum values of the axes `ALim` property (`AlphaDataMapping` set to `scaled`, the default).

`AlphaDataMapping` `none | direct | {scaled}`

Transparency mapping method. This property determines how MATLAB interprets indexed alpha data. This property can be any of the following:

- `none` - The transparency values of `AlphaData` are between 0 and 1 or are clamped to this range (the default).
- `scaled` - Transform the `AlphaData` to span the portion of the alphamap indicated by the axes `ALim` property, linearly mapping data values to alpha values.
- `direct` - use the `AlphaData` as indices directly into the alphamap. When not scaled, the data are usually integer values ranging from `1` to `length(alphamap)`. MATLAB maps values less than `1` to the first alpha

value in the alphamap, and values greater than `length(alphamap)` to the last alpha value in the alphamap. Values with a decimal portion are fixed to the nearest, lower integer. If `AlphaData` is an array `unit8` integers, then the indexing begins at `0` (i.e., MATLAB maps a value of `0` to the first alpha value in the alphamap).

`AmbientStrength` scalar >= 0 and <= 1

Strength of ambient light. This property sets the strength of the ambient light, which is a nondirectional light source that illuminates the entire scene. You must have at least one visible light object in the axes for the ambient light to be visible. The axes `AmbientLightColor` property sets the color of the ambient light, which is therefore the same on all objects in the axes.

You can also set the strength of the diffuse and specular contribution of light objects. See the surface `DiffuseStrength` and `SpecularStrength` properties.

`BackFaceLighting` `unlit | lit | reverselit`

Face lighting control. This property determines how faces are lit when their vertex normals point away from the camera.

- `unlit` – face is not lit
- `lit` – face lit in normal way
- `reverselit` – face is lit as if the vertex pointed towards the camera

This property is useful for discriminating between the internal and external surfaces of an object. See the *Using MATLAB Graphics* manual for an example.

`BusyAction` `cancel | {queue}`

Callback routine interruption. The `BusyAction` property enables you to control how MATLAB handles events that potentially interrupt executing callback routines. If there is a callback routine executing, subsequently invoked callback routines always attempt to interrupt it. If the `Interruptible` property of the object whose callback is executing is set to `on` (the default), then interruption occurs at the next point where the event queue is processed. If the `Interruptible` property is `off`, the `BusyAction` property (of the object owning the executing callback) determines how MATLAB handles the event. The choices are:

- `cancel` – discard the event that attempted to execute a second callback routine.

- `queue` – queue the event that attempted to execute a second callback routine until the current callback finishes.

ButtonDownFcn string

Button press callback routine. A callback routine that executes whenever you press a mouse button while the pointer is over the surface object. Define this routine as a string that is a valid MATLAB expression or the name of an M-file. The expression executes in the MATLAB workspace.

CData matrix

Vertex colors. A matrix containing values that specify the color at every point in `ZData`. If you set the `FaceColor` property to `texturemap`, `CData` does not need to be the same size as `ZData`. In this case, MATLAB maps `CData` to conform to the surface defined by `ZData`.

You can specify color as indexed values or true color. Indexed color data specifies a single value for each vertex. These values are either scaled to map linearly into the current colormap (see `caxis`) or interpreted directly as indices into the colormap, depending on the setting of the `CDataMapping` property.

True color defines an RGB value for each vertex. If the coordinate data (`XData` for example) are contained in *m*-by-*n* matrices, then `CData` must be an *m*-by-*n*-3 array. The first page contains the red components, the second the green components, and the third the blue components of the colors.

On computer displays that cannot display true color (e.g., 8-bit displays), MATLAB uses dithering to approximate the RGB triples using the colors in the figure's `Colormap` and `Dithermap`. By default, `Dithermap` uses the `colorcube(64)` colormap. You can also specify your own dithermap.

CDataMapping `{scaled} | direct`

Direct or scaled color mapping. This property determines how MATLAB interprets indexed color data used to color the surface. (If you use true color specification for `CData`, this property has no effect.)

- `scaled` – transform the color data to span the portion of the colormap indicated by the axes `CLim` property, linearly mapping data values to colors. See the caxis reference page for more information on this mapping.
- `direct` – use the color data as indices directly into the colormap. The color data should then be integer values ranging from 1 to `length(colormap)`.

MATLAB maps values less than 1 to the first color in the colormap, and values greater than `length(colormap)` to the last color in the colormap. Values with a decimal portion are fixed to the nearest, lower integer.

Children matrix of handles

Always the empty matrix; surface objects have no children.

Clipping `{on} | off`

Clipping to axes rectangle. When `Clipping` is `on`, MATLAB does not display any portion of the surface that is outside the axes rectangle.

CreateFcn string

Callback routine executed during object creation. This property defines a callback routine that executes when MATLAB creates a surface object. You must define this property as a default value for surfaces. For example, the statement,

```
set(0,'DefaultSurfaceCreateFcn',...
      'set(gcf,''DitherMap'',my_dithermap)')
```

defines a default value on the root level that sets the figure `DitherMap` property whenever you create a surface object. MATLAB executes this routine after setting all surface properties. Setting this property on an existing surface object has no effect.

The handle of the object whose `CreateFcn` is being executed is accessible only through the root `CallbackObject` property, which you can query using `gcbo`.

DeleteFcn string

Delete surface callback routine. A callback routine that executes when you delete the surface object (e.g., when you issue a `delete` command or clear the axes or figure). MATLAB executes the routine before destroying the object's properties so these values are available to the callback routine.

The handle of the object whose `DeleteFcn` is being executed is accessible only through the root `CallbackObject` property, which you can query using `gcbo`.

DiffuseStrength scalar >= 0 and <= 1

Intensity of diffuse light. This property sets the intensity of the diffuse component of the light falling on the surface. Diffuse light comes from light objects in the axes.

You can also set the intensity of the ambient and specular components of the light on the surface object. See the `AmbientStrength` and `SpecularStrength` properties.

EdgeAlpha `{scalar = 1} | flat | interp`

Transparency of the surface edges. This property can be any of the following:

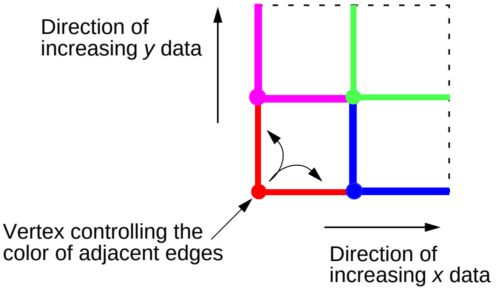
- `scalar` - A single non-Nan scalar value between `0` and `1` that controls the transparency of all the edges of the object. `1` (the default) is fully opaque and `0` means completely transparent.
- `flat` - The alpha data (`AlphaData`) value for the first vertex of the face determines the transparency of the edges.
- `interp` - Linear interpolation of the alpha data (`AlphaData`) values at each vertex determine the transparency of the edge.

Note that you must specify `AlphaData` as a matrix equal in size to `ZData` to use `flat` or `interp` `EdgeAlpha`.

EdgeColor `{ColorSpec} | none | flat | interp`

Color of the surface edge. This property determines how MATLAB colors the edges of the individual faces that make up the surface:

- `ColorSpec` — A three-element RGB vector or one of MATLAB's predefined names, specifying a single color for edges. The default `EdgeColor` is black. See `ColorSpec` for more information on specifying color.
- `none` — Edges are not drawn.
- `flat` — The `CData` value of the first vertex for a face determines the color of each edge.



- `interp` — Linear interpolation of the `CData` values at the face vertices determines the edge color.

`EdgeLighting` `{none} | flat | gouraud | phong`

Algorithm used for lighting calculations. This property selects the algorithm used to calculate the effect of light objects on surface edges. Choices are:

- `none` – Lights do not affect the edges of this object.
- `flat` – The effect of light objects is uniform across each edge of the surface.
- `gouraud` – The effect of light objects is calculated at the vertices and then linearly interpolated across the edge lines.
- `phong` – The effect of light objects is determined by interpolating the vertex normals across each edge line and calculating the reflectance at each pixel. Phong lighting generally produces better results than Gouraud lighting, but takes longer to render.

`EraseMode` `{normal} | none | xor | background`

Erase mode. This property controls the technique MATLAB uses to draw and erase surface objects. Alternative erase modes are useful for creating animated sequences, where control of the way individual objects redraw is necessary to improve performance and obtain the desired effect.

- `normal` — Redraw the affected region of the display, performing the three-dimensional analysis necessary to ensure that all objects are rendered correctly. This mode produces the most accurate picture, but is the slowest. The other modes are faster, but do not perform a complete redraw and are therefore less accurate.
- `none` — Do not erase the surface when it is moved or destroyed. While the object is still visible on the screen after erasing with `EraseMode` `none`, you cannot print it because MATLAB stores no information about its former location.
- `xor` — Draw and erase the surface by performing an exclusive OR (XOR) with each pixel index of the screen behind it. Erasing the surface does not damage the color of the objects behind it. However, surface color depends on the color of the screen behind it and is correctly colored only when over the axes background `Color`, or the figure background `Color` if the axes `Color` is set to `none`.

- `background` — Erase the surface by drawing it in the axes' background `Color`, or the figure background `Color` if the axes `Color` is set to none. This damages objects that are behind the erased object, but surface objects are always properly colored.

Printing with Non-normal Erase Modes. MATLAB always prints figures as if the `EraseMode` of all objects is `normal`. This means graphics objects created with `EraseMode` set to `none`, `xor`, or `background` can look different on screen than on paper. On screen, MATLAB may mathematically combine layers of colors (e.g., XORing a pixel color with that of the pixel behind it) and ignore three-dimensional sorting to obtain greater rendering speed. However, these techniques are not applied to the printed output.

You can use the MATLAB `getframe` command or other screen capture application to create an image of a figure containing non-normal mode objects.

FaceAlpha `{scalar = 1} | flat | interp | texturemap`

Transparency of the surface faces. This property can be any of the following:

- `scalar` - A single non-NaN scalar value between `0` and `1` that controls the transparency of all the faces of the object. `1` (the default) is fully opaque and `0` is completely transparent (invisible).
- `flat` - The values of the alpha data (`AlphaData`) determine the transparency for each face. The alpha data at the first vertex determines the transparency of the entire face.
- `interp` - Bilinear interpolation of the alpha data (`AlphaData`) at each vertex determine the transparency of each face.
- `texturemap` – Use transparency for the texturemap.

Note that you must specify `AlphaData` as a matrix equal in size to `ZData` to use `flat` or `interp` `FaceAlpha`.

FaceColor `ColorSpec | none | {flat} | interp`

Color of the surface face. This property can be any of the following:

- `ColorSpec` — A three-element RGB vector or one of MATLAB's predefined names, specifying a single color for faces. See `ColorSpec` for more information on specifying color.
- `none` — Do not draw faces. Note that edges are drawn independently of faces.

- `flat` — The values of `CData` determine the color for each face of the surface. The color data at the first vertex determines the color of the entire face.
- `interp` — Bilinear interpolation of the values at each vertex (the `CData`) determines the coloring of each face.
- `texturemap` — Texture map the `CData` to the surface. MATLAB transforms the color data so that it conforms to the surface. (See the texture mapping example.)

`FaceLighting` `{none} | flat | gouraud | phong`

Algorithm used for lighting calculations. This property selects the algorithm used to calculate the effect of light objects on the surface. Choices are:

- `none` – Lights do not affect the faces of this object.
- `flat` – The effect of light objects is uniform across the faces of the surface. Select this choice to view faceted objects.
- `gouraud` – The effect of light objects is calculated at the vertices and then linearly interpolated across the faces. Select this choice to view curved surfaces.
- `phong` – The effect of light objects is determined by interpolating the vertex normals across each face and calculating the reflectance at each pixel. Select this choice to view curved surfaces. Phong lighting generally produces better results than Gouraud lighting, but takes longer to render.

`HandleVisibility` `{on} | callback | off`

Control access to object's handle by command-line users and GUIs. This property determines when an object's handle is visible in its parent's list of children. This property is useful for preventing command-line users from accidentally drawing into or deleting a figure that contains only user interface devices (such as a dialog box).

Handles are always visible when `HandleVisibility` is `on`.

Setting `HandleVisibility` to `callback` causes handles to be visible from within callback routines or functions invoked by callback routines, but not from within functions invoked from the command line. This provides a means to protect GUIs from command-line users, while allowing callback routines to have complete access to object handles.

Setting `HandleVisibility` to `off` makes handles invisible at all times. This may be necessary when a callback routine invokes a function that might potentially damage the GUI (such as evaluating a user-typed string), and so temporarily hides its own handles during the execution of that function.

When a handle is not visible in its parent's list of children, it cannot be returned by functions that obtain handles by searching the object hierarchy or querying handle properties. This includes `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close`.

When a handle's visibility is restricted using `callback` or `off`, the object's handle does not appear in its parent's `Children` property, figures do not appear in the root's `CurrentFigure` property, objects do not appear in the root's `CallbackObject` property or in the figure's `CurrentObject` property, and axes do not appear in their parent's `CurrentAxes` property.

You can set the root `ShowHiddenHandles` property to `on` to make all handles visible, regardless of their `HandleVisibility` settings (this does not affect the values of the `HandleVisibility` properties).

Handles that are hidden are still valid. If you know an object's handle, you can `set` and `get` its properties, and pass it to any function that operates on handles.

`HitTest` `{on} | off`

Selectable by mouse click. `HitTest` determines if the surface can become the current object (as returned by the `gco` command and the figure `CurrentObject` property) as a result of a mouse click on the surface. If `HitTest` is `off`, clicking on the surface selects the object below it (which maybe the axes containing it).

`Interruptible` `{on} | off`

Callback routine interruption mode. The `Interruptible` property controls whether a surface callback routine can be interrupted by subsequently invoked callback routines. Only callback routines defined for the `ButtonDownFcn` are affected by the `Interruptible` property. MATLAB checks for events that can interrupt a callback routine only when it encounters a `drawnow`, `figure`, `getframe`, or `pause` command in the routine. See the `BusyAction` property for related information.

LineStyle {-} | -- | : | -. | none

Edge line type. This property determines the line style used to draw surface edges. The available line styles are shown in this table.

Symbol	Line Style
–	solid line (default)
––	dashed line
:	dotted line
–.	dash-dot line
none	no line

LineWidth scalar

Edge line width. The width of the lines in points used to draw surface edges. The default width is 0.5 points (1 point = 1/72 inch).

Marker marker symbol (see table)

Marker symbol. The Marker property specifies symbols that display at vertices. You can set values for the Marker property independently from the LineStyle property.

You can specify these markers.

Marker Specifier	Description
+	plus sign
o	circle
*	asterisk
.	point
x	cross
s	square
d	diamond

Marker Specifier	Description
`^`	upward pointing triangle
`v`	downward pointing triangle
`>`	right pointing triangle
`<`	left pointing triangle
`p`	five-pointed star (pentagram)
`h`	six-pointed star (hexagram)
none	no marker (default)

`MarkerEdgeColor` `ColorSpec | none | {auto}`

Marker edge color. The color of the marker or the edge color for filled markers (circle, square, diamond, pentagram, hexagram, and the four triangles).

- `ColorSpec` defines a single color to use for the edge (see `ColorSpec` for more information).
- `none` specifies no color, which makes nonfilled markers invisible.
- `auto` uses the same color as the `EdgeColor` property.

`MarkerFaceColor` `ColorSpec | {none} | auto`

Marker face color. The fill color for markers that are closed shapes (circle, square, diamond, pentagram, hexagram, and the four triangles).

- `ColorSpec` defines a single color to use for all marker on the surface (see `ColorSpec` for more information).
- `none` makes the interior of the marker transparent, allowing the background to show through.
- `auto` uses the `CData` for the vertex located by the marker to determine the color.

`MarkerSize` size in points

Marker size. A scalar specifying the marker size, in points. The default value for `MarkerSize` is six points (1 point = 1/72 inch). Note that MATLAB draws the point marker at 1/3 the specified marker size.

MeshStyle `{both} | row | column`

Row and column lines. This property specifies whether to draw all edge lines or just row or column edge lines.

- `both` draws edges for both rows and columns.
- `row` draws row edges only.
- `column` draws column edges only.

NormalMode `{auto} | manual`

MATLAB -generated or user-specified normal vectors. When this property is `auto`, MATLAB calculates vertex normals based on the coordinate data. If you specify your own vertex normals, MATLAB sets this property to `manual` and does not generate its own data. See also the `VertexNormals` property.

Parent handle

Surface's parent object. The parent of a surface object is the axes in which it is displayed. You can move a surface object to another axes by setting this property to the handle of the new parent.

Selected `on | {off}`

Is object selected? When this property is `on`, MATLAB displays a dashed bounding box around the surface if the `SelectionHighlight` property is also `on`. You can, for example, define the `ButtonDownFcn` to set this property, allowing users to select the object with the mouse.

SelectionHighlight `{on} | off`

Objects highlight when selected. When the `Selected` property is `on`, MATLAB indicates the selected state by drawing a dashed bounding box around the surface. When `SelectionHighlight` is `off`, MATLAB does not draw the handles.

SpecularColorReflectance scalar in the range 0 to 1

Color of specularly reflected light. When this property is 0, the color of the specularly reflected light depends on both the color of the object from which it reflects and the color of the light source. When set to 1, the color of the specularly reflected light depends only on the color or the light source (i.e., the light object `Color` property). The proportions vary linearly for values in between.

SpecularExponent scalar >= 1

Harshness of specular reflection. This property controls the size of the specular spot. Most materials have exponents in the range of 5 to 20.

SpecularStrength scalar >= 0 and <= 1

Intensity of specular light. This property sets the intensity of the specular component of the light falling on the surface. Specular light comes from light objects in the axes.

You can also set the intensity of the ambient and diffuse components of the light on the surface object. See the `AmbientStrength` and `DiffuseStrength` properties. Also see the `material` function.

Tag string

User-specified object label. The `Tag` property provides a means to identify graphics objects with a user-specified label. This is particularly useful when constructing interactive graphics programs that would otherwise need to define object handles as global variables or pass them as arguments between callback routines. You can define `Tag` as any string.

Type string (read only)

Class of the graphics object. The class of the graphics object. For surface objects, `Type` is always the string `'surface'`.

UIContextMenu handle of a uicontextmenu object

Associate a context menu with the surface. Assign this property the handle of a uicontextmenu object created in the same figure as the surface. Use the `uicontextmenu` function to create the context menu. MATLAB displays the context menu whenever you right-click over the surface.

UserData matrix

User-specified data. Any matrix you want to associate with the surface object. MATLAB does not use this data, but you can access it using the `set` and `get` commands.

VertexNormals vector or matrix

Surface normal vectors. This property contains the vertex normals for the surface. MATLAB generates this data to perform lighting calculations. You can supply your own vertex normal data, even if it does not match the coordinate data. This can be useful to produce interesting lighting effects.

Visible `{on} | off`

Surface object visibility. By default, all surfaces are visible. When set to `off`, the surface is not visible, but still exists and you can query and set its properties.

XData vector or matrix

X-coordinates. The *x*-position of the surface points. If you specify a row vector, `surface` replicates the row internally until it has the same number of columns as `ZData`.

YData vector or matrix

Y-coordinates. The *y*-position of the surface points. If you specify a row vector, `surface` replicates the row internally until it has the same number of rows as `ZData`.

ZData matrix

Z-coordinates. *Z*-position of the surface points. See the Description section for more information.

Purpose

Surface plot with colormap-based lighting

Syntax

```
surfl(Z)
surfl(X,Y,Z)
surfl(...,'light')
surfl(...,s)
surfl(X,Y,Z,s,k)
h = surfl(...)
```

Description

The `surfl` function displays a shaded surface based on a combination of ambient, diffuse, and specular lighting models.

`surfl(Z)` and `surfl(X,Y,Z)` create three-dimensional shaded surfaces using the default direction for the light source and the default lighting coefficients for the shading model. `X`, `Y`, and `Z` are vectors or matrices that define the *x*, *y*, and *z* components of a surface.

`surfl(...,'light')` produces a colored, lighted surface using a MATLAB light object. This produces results different from the default lighting method, surfl(...,'cdata'), which changes the color data for the surface to be the reflectance of the surface.

`surfl(...,s)` specifies the direction of the light source. `s` is a two- or three-element vector that specifies the direction from a surface to a light source. `s = [sx sy sz]` or `s = [azimuth elevation]`. The default `s` is 45° counterclockwise from the current view direction.

`surfl(X,Y,Z,s,k)` specifies the reflectance constant. `k` is a four-element vector defining the relative contributions of ambient light, diffuse reflection, specular reflection, and the specular shine coefficient. `k = [ka kd ks shine]` and defaults to `[.55,.6,.4,10]`.

`h = surfl(...)` returns a handle to a surface graphics object.

Remarks

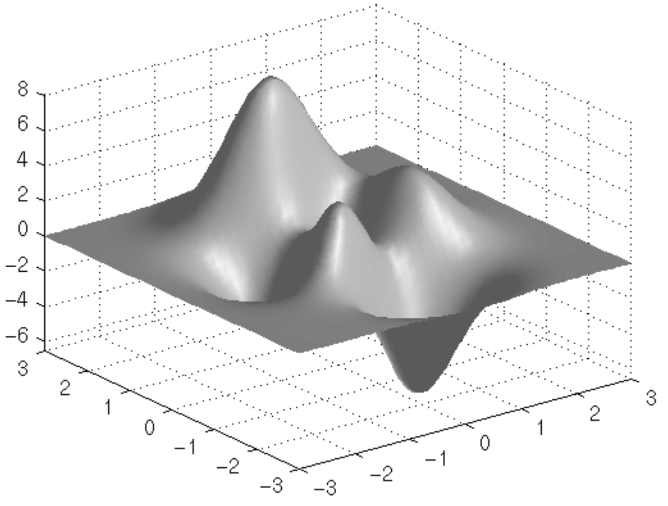
For smoother color transitions, use colormaps that have linear intensity variations (e.g., `gray`, `copper`, `bone`, `pink`).

The ordering of points in the `X`, `Y`, and `Z` matrices define the inside and outside of parametric surfaces. If you want the opposite side of the surface to reflect the light source, use `surfl(X',Y',Z')`. Because of the way surface normal vectors are computed, `surfl` requires matrices that are at least 3-by-3.

Examples

View peaks using colormap-based lighting.

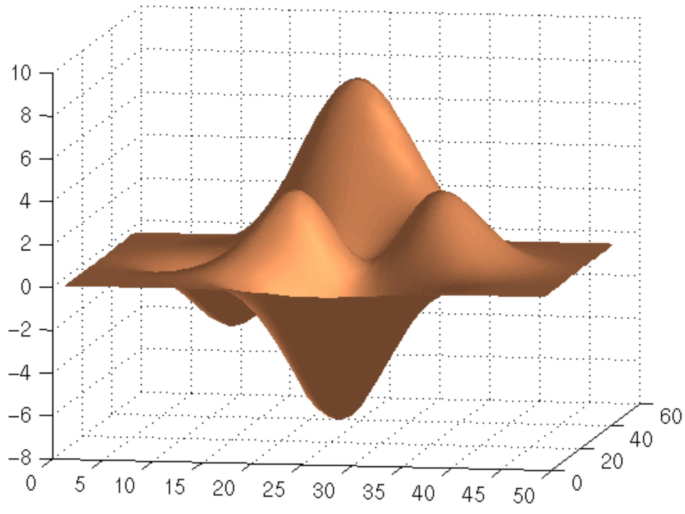
```
[x,y] = meshgrid(–3:1/8:3);
z = peaks(x,y);
surfl(x,y,z);
shading interp
colormap(gray);
axis([–3  3  –3  3  –8  8])
```



To plot a lighted surface from a view direction other than the default.

```
view([10 10])
grid on
hold on
surfl(peaks)
shading interp
colormap copper
```

```
hold off
```



See Also colormap, shading, light

surfnorm

Purpose Compute and display 3-D surface normals

Syntax

```
surfnorm(Z)
surfnorm(X,Y,Z)
[Nx,Ny,Nz] = surfnorm(...)
```

Description The `surfnorm` function computes surface normals for the surface defined by `X`, `Y`, and `Z`. The surface normals are unnormalized and valid at each vertex. Normals are not shown for surface elements that face away from the viewer.

`surfnorm(Z)` and `surfnorm(X,Y,Z)` plot a surface and its surface normals. `Z` is a matrix that defines the z component of the surface. `X` and `Y` are vectors or matrices that define the *x* and *y* components of the surface.

`[Nx,Ny,Nz] = surfnorm(...)` returns the components of the three-dimensional surface normals for the surface.

Remarks The direction of the normals is reversed by calling `surfnorm` with transposed arguments:

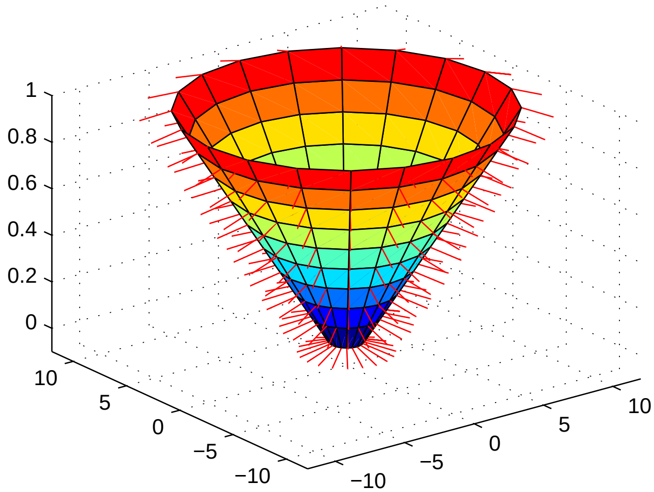
```
surfnorm(X',Y',Z')
```

`surfl` uses `surfnorm` to compute surface normals when calculating the reflectance of a surface.

Algorithm The surface normals are based on a bicubic fit of the data in `X`, `Y`, and `Z`. For each vertex, diagonal vectors are computed and crossed to form the normal.

Examples Plot the normal vectors for a truncated cone.

```
[x,y,z] = cylinder(1:10);
surfnorm(x,y,z)
axis([–12 12 –12 12 –0.1 1])
```



See Also `surf`, `quiver3`

Purpose Singular value decomposition

Syntax

```
s = svd(X)
[U,S,V] = svd(X)
[U,S,V] = svd(X,0)
```

Description The svd command computes the matrix singular value decomposition.

s = svd(X) returns a vector of singular values.

[U,S,V] = svd(X) produces a diagonal matrix S of the same dimension as X, with nonnegative diagonal elements in decreasing order, and unitary matrices U and V so that X = U*S*V'.

[U,S,V] = svd(X,0) produces the "economy size" decomposition. If X is m-by-n with m > n, then svd computes only the first n columns of U and S is n-by-n.

Examples For the matrix

```
X =
     1    2
     3    4
     5    6
     7    8
```

the statement

```
[U,S,V] = svd(X)
```

produces

```
U =
    -0.1525   -0.8226   -0.3945   -0.3800
    -0.3499   -0.4214    0.2428    0.8007
    -0.5474   -0.0201    0.6979   -0.4614
    -0.7448    0.3812   -0.5462    0.0407

S =
     14.2691         0
           0    0.6268
```

```
              0         0
              0         0

V =
    -0.6414     0.7672
    -0.7672    -0.6414
```

The economy size decomposition generated by

```
[U,S,V] = svd(X,0)
```

produces

```
U =
    -0.1525   -0.8226
    -0.3499   -0.4214
    -0.5474   -0.0201
    -0.7448    0.3812
S =
    14.2691         0
          0    0.6268
V =
    -0.6414    0.7672
    -0.7672   -0.6414
```

Algorithm

svd uses LAPACK routines to compute the singular value decomposition.

Matrix	Routine
Real	DGESVD
Complex	ZGESVD

Diagnostics

If the limit of 75 QR step iterations is exhausted while seeking a singular value, this message appears:

```
Solution will not converge.
```

References

[1] Anderson, E., Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen,

LAPACK User's Guide (http://www.netlib.org/lapack/lug/lapack_lug.html), Third Edition, SIAM, Philadelphia, 1999.

Purpose

A few singular values

Syntax

```
s = svds(A)
s = svds(A,k)
s = svds(A,k,0)
[U,S,V] = svds(A,...)
```

Description

`svds(A)` computes the five largest singular values and associated singular vectors of the matrix `A`.

`svds(A,k)` computes the `k` largest singular values and associated singular vectors of the matrix `A`.

`svds(A,k,0)` computes the `k` smallest singular values and associated singular vectors.

With one output argument, `s` is a vector of singular values. With three output arguments and if `A` is `m`-by-`n`:

- `U` is `m`-by-`k` with orthonormal columns
- `S` is `k`-by-`k` diagonal
- `V` is `n`-by-`k` with orthonormal columns
- `U*S*V'` is the closest rank `k` approximation to `A`

Algorithm

`svds(A,k)` uses `eigs` to find the `k` largest magnitude eigenvalues and corresponding eigenvectors of `B = [0 A; A' 0]`.

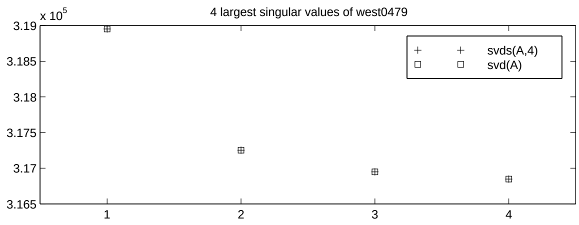
`svds(A,k,0)` uses `eigs` to find the `2k` smallest magnitude eigenvalues and corresponding eigenvectors of `B = [0 A; A' 0]`, and then selects the `k` positive eigenvalues and their eigenvectors.

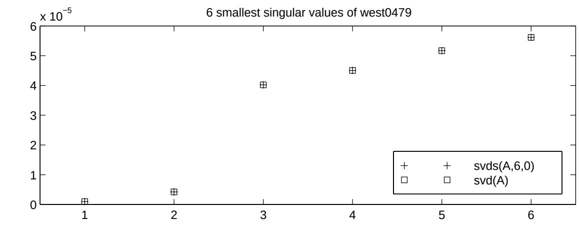
Example

`west0479` is a real 479-by-479 sparse matrix. `svd` calculates all 479 singular values. `svds` picks out the largest and smallest singular values.

```
load west0479
s = svd(full(west0479))
sl = svds(west0479,4)
ss = svds(west0479,6,0)
```

These plots show some of the singular values of `west0479` as computed by `svd` and `svds`.





The largest singular value of `west0479` can be computed a few different ways:

```
svds(west0479,1) =
 3.189517598808622e+05

max(svd(full(west0479))) =
 3.18951759880862e+05

norm(full(west0479)) =
 3.189517598808623e+05
```

and estimated:

```
normest(west0479) =
 3.189385666549991e+05
```

See Also `svd`, `eigs`

Purpose Switch among several cases based on expression

Syntax

```
switch switch_expr
  case case_expr
    statement,...,statement
  case {case_expr1,case_expr2,case_expr3,...}
    statement,...,statement
...
  otherwise
    statement,...,statement
end
```

Discussion The `switch` statement syntax is a means of conditionally executing code. In particular, `switch` executes one set of statements selected from an arbitrary number of alternatives. Each alternative is called a *case*, and consists of:

- The case statement
- One or more case expressions
- One or more statements

In its basic syntax, `switch` executes the statements associated with the first case where *switch_expr == case_expr*. When the case expression is a cell array (as in the second case above), the `case_expr` matches if any of the elements of the cell array match the switch expression. If no case expression matches the switch expression, then control passes to the `otherwise` case (if it exists). After the case is executed, program execution resumes with the statement after the `end`.

The `switch_expr` can be a scalar or a string. A scalar `switch_expr` matches a `case_expr` if `switch_expr==case_expr`. A string `switch_expr` matches a `case_expr` if `strcmp(switch_expr,case_expr)` returns `1` (true).

Note for C Programmers Unlike the C language `switch` construct, MATLAB's `switch` does not "fall through." That is, `switch` executes only the first matching case, subsequent matching cases do not execute. Therefore, `break` statements are not used.

Examples

To execute a certain block of code based on what the string, `method`, is set to,

```
method = 'Bilinear';

switch lower(method)
   case {'linear','bilinear'}
      disp('Method is linear')
   case 'cubic'
      disp('Method is cubic')
   case 'nearest'
      disp('Method is nearest')
   otherwise
      disp('Unknown method.')
end

Method is linear
```

See Also

`case`, `end`, `if`, `otherwise`, `while`

Purpose Symmetric approximate minimum degree permutation

Syntax

```
p = symamd(S)
p = symamd(S,knobs)
[p,stats] = symamd(S)
[p,stats] = symamd(S,knobs)
```

Description `p = symamd(S)` for a symmetric positive definite matrix `S`, returns the permutation vector `p` such that `S(p,p)` tends to have a sparser Cholesky factor than `S`. To find the ordering for `S`, `symamd` constructs a matrix `M` such that `spones(M'*M) = spones (S)`, and then computes `p = colamd(M)`. The `symamd` function may also work well for symmetric indefinite matrices.

`S` must be square; only the strictly lower triangular part is referenced.

`knobs` is a scalar. If `S` is `n`-by-`n`, rows and columns with more than `knobs*n` entries are removed prior to ordering, and ordered last in the output permutation `p`. If the `knobs` parameter is not present, then `knobs = spparms('wh_frac')`.

`stats` is an optional vector that provides data about the ordering and the validity of the matrix `S`.

`stats(1)`	Number of dense or empty rows ignored by `symamd`
`stats(2)`	Number of dense or empty columns ignored by `symamd`
`stats(3)`	Number of garbage collections performed on the internal data structure used by `symamd` (roughly of size `8.4*nnz(tril(S,-1)) + 9n` integers)
`stats(4)`	`0` if the matrix is valid, or `1` if invalid
`stats(5)`	Rightmost column index that is unsorted or contains duplicate entries, or `0` if no such column exists
`stats(6)`	Last seen duplicate or out-of-order row index in the column index given by `stats(5)`, or `0` if no such row index exists
`stats(7)`	Number of duplicate and out-of-order row indices

Although, MATLAB built-in functions generate valid sparse matrices, a user may construct an invalid sparse matrix using the MATLAB C or Fortran APIs and pass it to `symamd`. For this reason, `symamd` verifies that `S` is valid:

- If a row index appears two or more times in the same column, `symamd` ignores the duplicate entries, continues processing, and provides information about the duplicate entries in `stats(4:7)`.
- If row indices in a column are out of order, `symamd` sorts each column of its internal copy of the matrix `S` (but does not repair the input matrix `S`), continues processing, and provides information about the out-of-order entries in `stats(4:7)`.
- If `S` is invalid in any other way, `symamd` cannot continue. It prints an error message, and returns no output arguments (`p` or `stats`) .

The ordering is followed by a symmetric elimination tree post-ordering.

Note `symamd` tends to be faster than `symmmd` and tends to return a better ordering.

See Also

`colamd`, `colmmd`, `colperm`, `spparms`, `symmmd`, `symrcm`

References

The authors of the code for `symamd` are Stefan I. Larimore and Timothy A. Davis (`davis@cise.ufl.edu`), University of Florida. The algorithm was developed in collaboration with John Gilbert, Xerox PARC, and Esmond Ng, Oak Ridge National Laboratory. Sparse Matrix Algorithms Research at the University of Florida: `http://www.cise.ufl.edu/research/sparse/`

Purpose Symbolic factorization analysis

Syntax

```
count = symbfact(A)
count = symbfact(A,'col')
count = symbfact(A,'sym')
[count,h,parent,post,R] = symbfact(...)
```

Description `count = symbfact(A)` returns the vector of row counts for the upper triangular Cholesky factor of a symmetric matrix whose upper triangle is that of `A`, assuming no cancellation during the factorization. `symbfact` should be much faster than `chol(A)`.

`count = symbfact(A,'col')` analyzes `A'*A` (without forming it explicitly).

`count = symbfact(A,'sym')` is the same as `count = symbfact(A)`.

`[count,h,parent,post,R] = symbfact(...)` has several optional return values.

`h`	Height of the elimination tree
`parent`	The elimination tree itself
`post`	Postordering permutation of the elimination tree
`R`	0-1 matrix whose structure is that of `chol(A)`

See Also `chol`, `etree`, `treelayout`

symmlq

Purpose Symmetric LQ method

Syntax

```
x = symmlq(A,b)
symmlq(A,b,tol)
symmlq(A,b,tol,maxit)
symmlq(A,b,tol,maxit,M)
symmlq(A,b,tol,maxit,M1,M2)
symmlq(A,b,tol,maxit,M1,M2,x0)
symmlq(afun,b,tol,maxit,m1fun,m2fun,x0,p1,p2,...)
[x,flag] = symmlq(A,b,...)
[x,flag,relres] = symmlq(A,b,...)
[x,flag,relres,iter] = symmlq(A,b,...)
[x,flag,relres,iter,resvec] = symmlq(A,b,...)
[x,flag,relres,iter,resvec,resveccg] = symmlq(A,b,...)
```

Description

`x = symmlq(A,b)` attempts to solve the system of linear equations `A*x=b` for `x`. The `n`-by-`n` coefficient matrix `A` must be symmetric but need not be positive definite. The column vector `b` must have length `n`. `A` can be a function `afun` such that `afun(x)` returns `A*x`.

If `symmlq` converges, a message to that effect is displayed. If `symmlq` fails to converge after the maximum number of iterations or halts for any reason, a warning message is printed displaying the relative residual `norm(b-A*x)/norm(b)` and the iteration number at which the method stopped or failed.

`symmlq(A,b,tol)` specifies the tolerance of the method. If `tol` is `[]`, then `symmlq` uses the default, `1e-6`.

`symmlq(A,b,tol,maxit)` specifies the maximum number of iterations. If `maxit` is `[]`, then `symmlq` uses the default, `min(n,20)`.

`symmlq(A,b,tol,maxit,M)` and `symmlq(A,b,tol,maxit,M1,M2)` use the symmetric positive definite preconditioner `M` or `M = M1*M2` and effectively solve the system `inv(sqrt(M))*A*inv(sqrt(M))*y = inv(sqrt(M))*b` for `y` and then return `x = inv(sqrt(M))*y`. If `M` is `[]` then `symmlq` applies no preconditioner. `M` can be a function that returns `M\x`.

`symmlq(A,b,tol,maxit,M1,M2,x0)` specifies the initial guess. If `x0` is `[]`, then `symmlq` uses the default, an all-zero vector.

`symmlq(afun,b,tol,maxit,m1fun,m2fun,x0,p1,p2,...)` passes parameters `p1,p2,...` to functions `afun(x,p1,p2,...)`, `m1fun(x,p1,p2,...)`, and `m2fun(x,p1,p2,...)`.

`[x,flag] = symmlq(A,b,tol,maxit,M1,M2,x0,p1,p2,...)` also returns a convergence flag.

Flag	Convergence
0	`symmlq` converged to the desired tolerance `tol` within `maxit` iterations.
1	`symmlq` iterated `maxit` times but did not converge.
2	Preconditioner `M` was ill-conditioned.
3	`symmlq` stagnated. (Two consecutive iterates were the same.)
4	One of the scalar quantities calculated during `symmlq` became too small or too large to continue computing.
5	Preconditioner `M` was not symmetric positive definite.

Whenever `flag` is not `0`, the solution `x` returned is that with minimal norm residual computed over all the iterations. No messages are displayed if the `flag` output is specified.

`[x,flag,relres] = symmlq(A,b,tol,maxit,M1,M2,x0,p1,p2,...)` also returns the relative residual `norm(b-A*x)/norm(b)`. If `flag` is `0`, `relres <= tol`.

`[x,flag,relres,iter] = symmlq(A,b,tol,maxit,M1,M2,x0,p1,p2,...)` also returns the iteration number at which `x` was computed, where `0 <= iter <= maxit`.

`[x,flag,relres,iter,resvec] = symmlq(A,b,tol,maxit,M1,M2,x0,p1,p2,...)` also returns a vector of estimates of the `symmlq` residual norms at each iteration, including `norm(b-A*x0)`.

`[x,flag,relres,iter,resvec,resveccg] = symmlq(A,b,tol,maxit,M1,M2,x0,p1,p2,...)` also returns a vector of estimates of the conjugate gradients residual norms at each iteration.

Examples

Example 1.

```
n = 100;
on = ones(n,1);
A = spdiags([-2*on 4*on -2*on],-1:1,n,n);
b = sum(A,2);
tol = 1e-10;
maxit = 50; M1 = spdiags(4*on,0,n,n);

x = symmlq(A,b,tol,maxit,M1,[],[]);
symmlq converged at iteration 49 to a solution with relative
residual 4.3e-015
```

Alternatively, use this matrix-vector product function

```
function y = afun(x,n)
   y = 4 * x;
   y(2:n) = y(2:n) - 2 * x(1:n-1);
   y(1:n-1) = y(1:n-1) - 2 * x(2:n);
```

as input to `symmlq`.

```
x1 = symmlq(@afun,b,tol,maxit,M1,[],[],n);
```

Example 2.

Use a symmetric indefinite matrix that fails with `pcg`.

```
A = diag([20:-1:1,-1:-1:-20]);
b = sum(A,2);      % The true solution is the vector of all ones.
x = pcg(A,b);      % Errors out at the first iteration.
pcg stopped at iteration 1 without converging to the desired
tolerance 1e-006 because a scalar quantity became too small or
too large to continue computing.
The iterate returned (number 0) has relative residual 1
```

However, `symmlq` can handle the indefinite matrix `A`.

```
x = symmlq(A,b,1e-6,40);
```

```
symmlq converged at iteration 39 to a solution with relative
residual 1.3e-007
```

See Also

`bicg`, `bicgstab`, `cgs`, `lsqr`, `gmres`, `minres`, `pcg`, `qmr`

`@` (function handle), `/` (slash)

References

[1] Barrett, R., M. Berry, T. F. Chan, et al., *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*, SIAM, Philadelphia, 1994.

[2] Paige, C. C. and M. A., “Solution of Sparse Indefinite Systems of Linear Equations.” *SIAM J. Numer. Anal.*, Vol.12, 1975, pp. 617-629.

Purpose Sparse symmetric minimum degree ordering

Syntax

```
p = symmmd(S)
```

Description `p = symmmd(S)` returns a symmetric minimum degree ordering of `S`. For a symmetric positive definite matrix `S`, this is a permutation `p` such that `S(p,p)` tends to have a sparser Cholesky factor than `S`. Sometimes `symmmd` works well for symmetric indefinite matrices too.

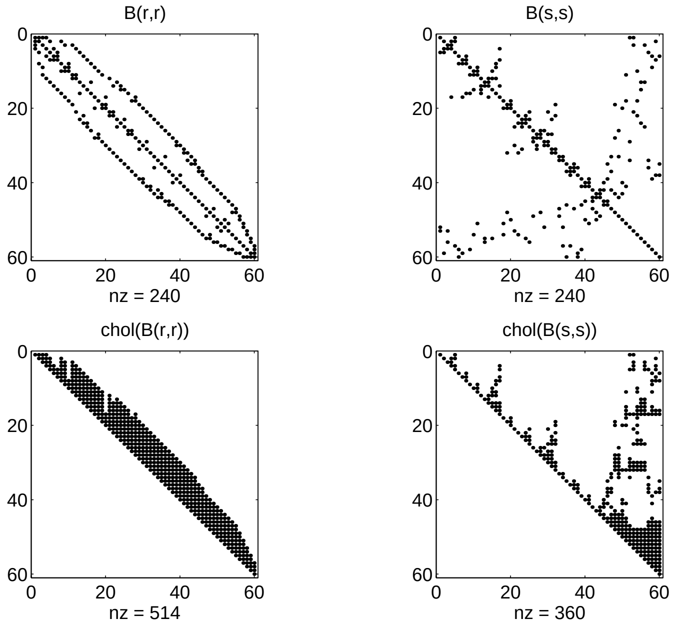
Remarks The minimum degree ordering is automatically used by `\` and `/` for the solution of symmetric, positive definite, sparse linear systems.

Some options and parameters associated with heuristics in the algorithm can be changed with `spparms`.

Algorithm The symmetric minimum degree algorithm is based on the column minimum degree algorithm. In fact, `symmmd(A)` just creates a nonzero structure `K` such that `K'*K` has the same nonzero structure as `A` and then calls the column minimum degree code for `K`.

Examples Here is a comparison of reverse Cuthill-McKee and minimum degree on the Bucky ball example mentioned in the `symrcm` reference page.

```
B = bucky+4*speye(60);
r = symrcm(B);
p = symmmd(B);
R = B(r,r);
S = B(p,p);
subplot(2,2,1), spy(R), title('B(r,r)')
subplot(2,2,2), spy(S), title('B(s,s)')
subplot(2,2,3), spy(chol(R)), title('chol(B(r,r))')
subplot(2,2,4), spy(chol(S)), title('chol(B(s,s))')
```



Even though this is a very small problem, the behavior of both orderings is typical. RCM produces a matrix with a narrow bandwidth which fills in almost completely during the Cholesky factorization. Minimum degree produces a structure with large blocks of contiguous zeros which do not fill in during the factorization. Consequently, the minimum degree ordering requires less time and storage for the factorization.

See Also

`colamd`, `colmmd`, `colperm`, `symamd`, `symrcm`

References

[1] Gilbert, John R., Cleve Moler, and Robert Schreiber, “Sparse Matrices in MATLAB: Design and Implementation,” *SIAM Journal on Matrix Analysis and Applications 13*, 1992, pp. 333-356.

symrcm

Purpose Sparse reverse Cuthill-McKee ordering

Syntax

```
r = symrcm(S)
```

Description `r = symrcm(S)` returns the symmetric reverse Cuthill-McKee ordering of `S`. This is a permutation `r` such that `S(r,r)` tends to have its nonzero elements closer to the diagonal. This is a good preordering for LU or Cholesky factorization of matrices that come from long, skinny problems. The ordering works for both symmetric and nonsymmetric `S`.

For a real, symmetric sparse matrix, `S`, the eigenvalues of `S(r,r)` are the same as those of `S`, but `eig(S(r,r))` probably takes less time to compute than `eig(S)`.

Algorithm The algorithm first finds a pseudoperipheral vertex of the graph of the matrix. It then generates a level structure by breadth-first search and orders the vertices by decreasing distance from the pseudoperipheral vertex. The implementation is based closely on the SPARSPAK implementation described by George and Liu.

Examples The statement

```
B = bucky
```

uses an M-file in the `demos` toolbox to generate the adjacency graph of a truncated icosahedron. This is better known as a soccer ball, a Buckminster Fuller geodesic dome (hence the name `bucky`), or, more recently, as a 60-atom carbon molecule. There are 60 vertices. The vertices have been ordered by numbering half of them from one hemisphere, pentagon by pentagon; then reflecting into the other hemisphere and gluing the two halves together. With this numbering, the matrix does not have a particularly narrow bandwidth, as the first spy plot shows

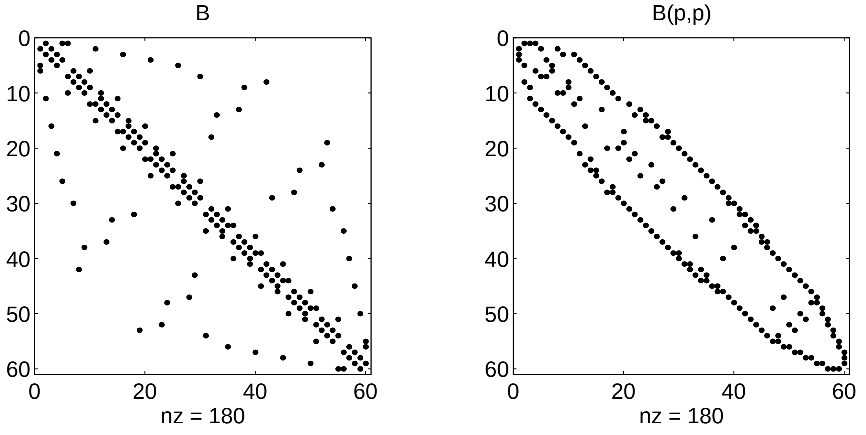
```
subplot(1,2,1), spy(B), title('B')
```

The reverse Cuthill-McKee ordering is obtained with

```
p = symrcm(B);
R = B(p,p);
```

The spy plot shows a much narrower bandwidth.

```
subplot(1,2,2), spy(R), title('B(p,p)')
```



This example is continued in the reference pages for `symmmd`.

The bandwidth can also be computed with

```
[i,j] = find(B);
bw = max(i-j) + 1
```

The bandwidths of `B` and `R` are 35 and 12, respectively.

See Also

`colamd`, `colmmd`, `colperm`, `symamd`, `symmmd`

References

[1] George, Alan and Joseph Liu, *Computer Solution of Large Sparse Positive Definite Systems*, Prentice-Hall, 1981.

[2] Gilbert, John R., Cleve Moler, and Robert Schreiber, “Sparse Matrices in MATLAB: Design and Implementation,” to appear in *SIAM Journal on Matrix Analysis*, 1992. A slightly expanded version is also available as a technical report from the Xerox Palo Alto Research Center.

symvar

Purpose Determine the symbolic variables in an expression

Syntax

```
symvar 'expr'
s = symvar('expr')
```

Description `symvar 'expr'` searches the expression, `expr`, for identifiers other than `i`, `j`, `pi`, `inf`, `nan`, `eps`, and common functions. `symvar` displays those variables that it finds or, if no such variable exists, displays an empty cell array, `{}`.

`s = symvar('expr')` returns the variables in a cell array of strings, `s`. If no such variable exists, `s` is an empty cell array.

Examples `symvar` finds variables `beta1` and `x`, but skips `pi` and the `cos` function.

```
symvar 'cos(pi*x - beta1)'

ans =

    'beta1'
    'x'
```

See Also `findstr`

Purpose

Tangent and hyperbolic tangent

Syntax

```
Y = tan(X)
Y = tanh(X)
```

Description

The `tan` and `tanh` functions operate element-wise on arrays. The functions' domains and ranges include complex values. All angles are in radians.

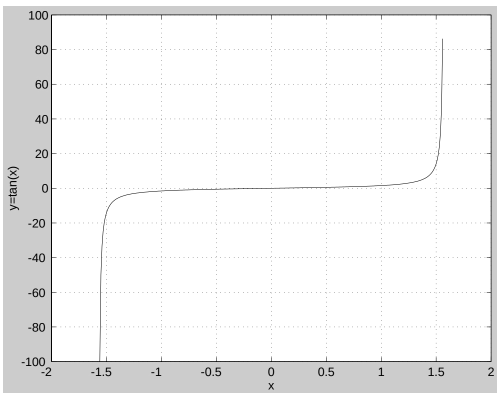
`Y = tan(X)` returns the circular tangent of each element of `X`.

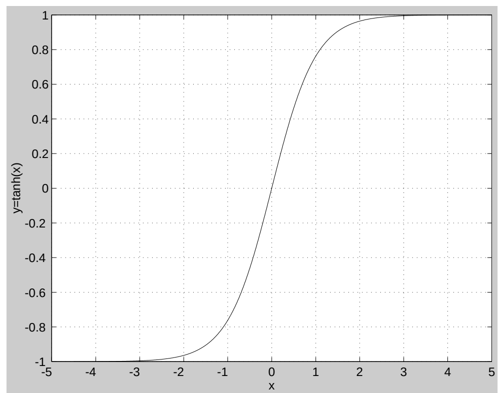
`Y = tanh(X)` returns the hyperbolic tangent of each element of `X`.

Examples

Graph the tangent function over the domain $-\pi/2 < x < \pi/2$, and the hyperbolic tangent function over the domain $-5 \le x \le 5$.

```
x = (-pi/2)+0.01:0.01:(pi/2)-0.01; plot(x,tan(x))
x = -5:0.01:5; plot(x,tanh(x))
```





The expression `tan(pi/2)` does not evaluate as infinite but as the reciprocal of the floating point accuracy `eps` since `pi` is only a floating-point approximation to the exact value of π.

Algorithm

tan and tanh use these algorithms.

$$\tan(z) = \frac{\sin(z)}{\cos(z)}$$

$$\tanh(z) = \frac{\sinh(z)}{\cosh(z)}$$

See Also

atan, atan2

Purpose Return the name of the system's temporary directory

Syntax

```
tmp_dir = tempdir
```

Description `tmp_dir = tempdir` returns the name of the system's temporary directory, if one exists. This function does not create a new directory.

See Opening Temporary Files and Directories for more information.

See Also `tempname`

tempname

Purpose Unique name for temporary file

Syntax

```
tmp_nam = tempname
```

Description

`tmp_nam = tempname` returns a unique string, `tmp_nam`, suitable for use as a temporary filename.

Note The filename that `tempname` generates is not guaranteed to be unique; however, it is likely to be so.

See Opening Temporary Files and Directories for more information.

See Also `tempdir`

Purpose Set graphics terminal type

Syntax

```
terminal
terminal('type')
```

Description To add terminal-specific settings (e.g., escape characters, line length), edit the file `terminal.m`.

`terminal` displays a menu of graphics terminal types, prompts for a choice, then configures MATLAB to run on the specified terminal.

`terminal('type')` accepts a terminal type string. Valid `'type'` strings are shown in the table.

Type	Description
`tek401x`	Tektronix 4010/4014
`tek4100`	Tektronix 4100
`tek4105`	Tektronix 4105
`retro`	Retrographics card
`sg100`	Selanar Graphics 100
`sg200`	Selanar Graphics 200
`vt240tek`	VT240 & VT340 Tektronix mode
`ergo`	Ergo terminal
`graphon`	Graphon terminal
`citoh`	C.Itoh terminal
`xtermtek`	xterm, Tektronix graphics
`wyse`	Wyse WY-99GT
`kermit`	MS-DOS Kermit 2.23
`hp2647`	Hewlett-Packard 2647

Type	Description (Continued)
hds	Human Designed Systems

Purpose Tetrahedron mesh plot

Syntax

```
tetramesh(T,X,c)
tetramesh(T,X)
h = tetramesh(...)
tetramesh(...,'param','value','param','value'...)
```

Description `tetramesh(T,X,c)` displays the tetrahedrons defined in the `m`-by-4 matrix `T` as mesh. `T` is usually the output of `delaunayn`. A row of `T` contains indices into `X` of the vertices of a tetrahedron. `X` is an `n`-by-3 matrix, representing `n` points in 3 dimension. The tetrahedron colors are defined by the vector `C`, which is used as indices into the current colormap.

Note If `T` is the output of `delaunay3`, then `X` is the concatenation of the `delaunay3` input arguments `x`, `y`, `z` interpreted as column vectors, i.e., `X = [x(:) y(:) z(:)]`.

`tetramesh(T,X)` uses `C = 1:m` as the color for the `m` tetrahedrons. Each tetrahedron has a different color (modulo the number of colors available in the current colormap).

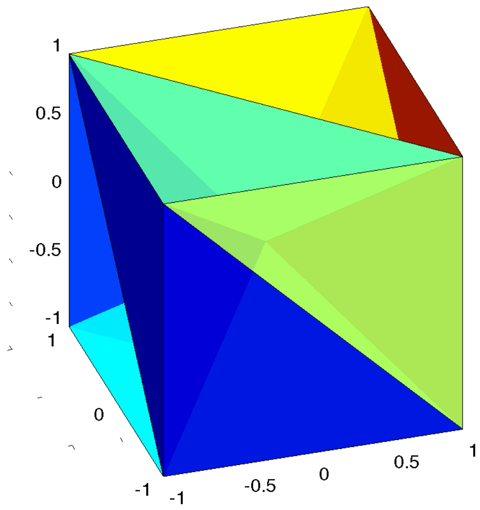
`h = tetramesh(...)` returns a vector of tetrahedron handles. Each element of `h` is a handle to the set of patches forming one tetrahedron. You can use these handles to view a particular tetrahedron by turning the patch `'Visible'` property `'on'` or `'off'`.

`tetramesh(...,'param','value','param','value'...)` allows additional patch property name/property value pairs to be used when displaying the tetrahedrons. For example, the default transparency parameter is set to `0.9`. You can overwrite this value by using the property name/property value pair `('FaceAlpha',value)` where `value` is a number between `0` and `1`. See `Patch Properties` for information about the available properties.

Examples Generate a 3-dimensional Delaunay tesselation, then use `tetramesh` to visualize the tetrahedrons that form the corresponding simplex.

```
d = [-1 1];
```

```
[x,y,z] = meshgrid(d,d,d);  % A cube
x = [x(:);0];
y = [y(:);0];
z = [z(:);0];
% [x,y,z] are corners of a cube plus the center.
X = [x(:) y(:) z(:)];
Tes = delaunayn(X)

Tes =
   9   1   5   6
   3   9   1   5
   2   9   1   6
   2   3   9   4
   2   3   9   1
   7   9   5   6
   7   3   9   5
   8   7   9   6
   8   2   9   6
   8   2   9   4
   8   3   9   4
   8   7   3   9

tetramesh(Tes,X);camorbit(20,0)
```



See Also delaunayn, patch, Patch Properties, trimesh, trisurf

texlabel

Purpose

Produce TeX format from character string

Syntax

```
texlabel(f)
texlabel(f,'literal')
```

Description

`texlabel(f)` converts the MATLAB expression `f` into the TeX equivalent for use in text strings. It processes Greek variable names (e.g., lambda, delta, etc.) into a string that displays as actual Greek letters.

`texlabel(f,'literal')` prints Greek variable names as literals.

If the string is too long to fit into a figure window, then the center of the expression is replaced with a tilde ellipsis (~~~).

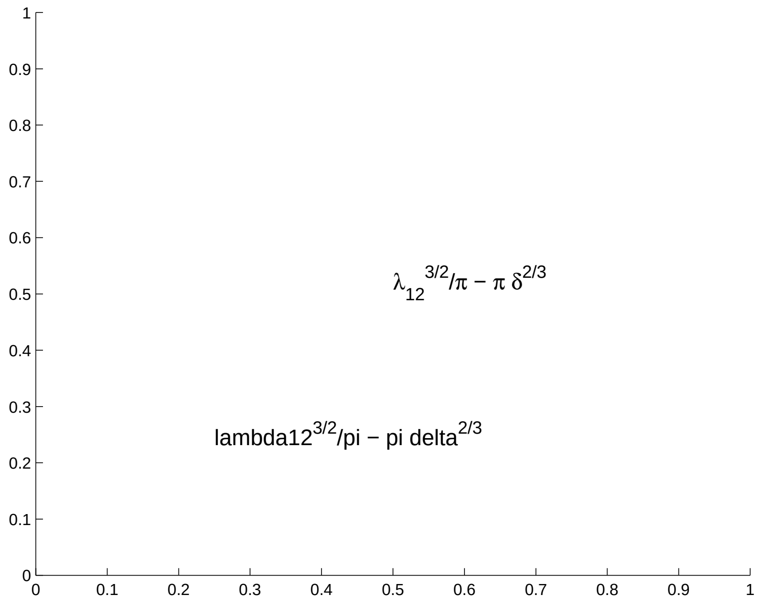
Examples

You can use `texlabel` as an argument to the `title`, `xlabel`, `ylabel`, `zlabel`, and `text` commands. For example,

```
title(texlabel('sin(sqrt(x^2 + y^2))/sqrt(x^2 + y^2)'))
```

By default, `texlabel` translates Greek variable names to the equivalent Greek letter. You can select literal interpretation by including the `literal` argument. For example, compare these two commands.

```
text(.5,.5,...
     texlabel('lambda12^(3/2)/pi - pi*delta^(2/3)'))
text(.25,.25,...
     texlabel('lambda12^(3/2)/pi - pi*delta^(2/3)','literal'))
```



See Also

`text`, `title`, `xlabel`, `ylabel`, `zlabel`, the text `String` property

Purpose Create text object in current axes

Syntax

```
text(x,y,'string')
text(x,y,z,'string')
text(...'PropertyName',PropertyValue...)
h = text(...)
```

Description `text` is the low-level function for creating text graphics objects. Use `text` to place character strings at specified locations.

`text(x,y,'string')` adds the string in quotes to the location specified by the point `(x,y)`.

`text(x,y,z,'string')` adds the string in 3-D coordinates.

`text(x,y,z,'string','PropertyName',PropertyValue....)` adds the string in quotes to location defined by the coordinates and uses the values for the specified text properties. See the text property list section at the end of this page for a list of text properties.

`text('PropertyName',PropertyValue....)` omits the coordinates entirely and specifies all properties using property name/property value pairs.

`h = text(..)` returns a column vector of handles to text objects, one handle per object. All forms of the `text` function optionally return this output argument.

See the `String` property for a list of symbols, including Greek letters.

Remarks Specify the text location coordinates (the `x`, `y`, and `z` arguments) in the data units of the current axes (see “Examples”). The `Extent`, `VerticalAlignment`, and `HorizontalAlignment` properties control the positioning of the character string with regard to the text location point.

If the coordinates are vectors, `text` writes the string at all locations defined by the list of points. If the character string is an array the same length as `x`, `y`, and `z`, `text` writes the corresponding row of the string array at each point specified.

When specifying strings for multiple text objects, the string can be

- a cell array of strings

- a padded string matrix
- a string vector using vertical slash characters ('|') as separators.

Each element of the specified string array creates a different text object.

When specifying the string for a single text object, cell arrays of strings and padded string matrices result in a text object with a multiline string, while vertical slash characters are not interpreted as separators and result in a single line string containing vertical slashes.

`text` is a low-level function that accepts property name/property value pairs as input arguments, however; the convenience form,

```
text(x,y,z,'string')
```

is equivalent to:

```
text('XData',x,'YData',y,'ZData',z,'String','string')
```

You can specify other properties only as property name/property value pairs. See the text property list at the end of this page for a description of each property. You can specify properties as property name/property value pairs, structure arrays, and cell arrays (see the `set` and `get` reference pages for examples of how to specify these data types).

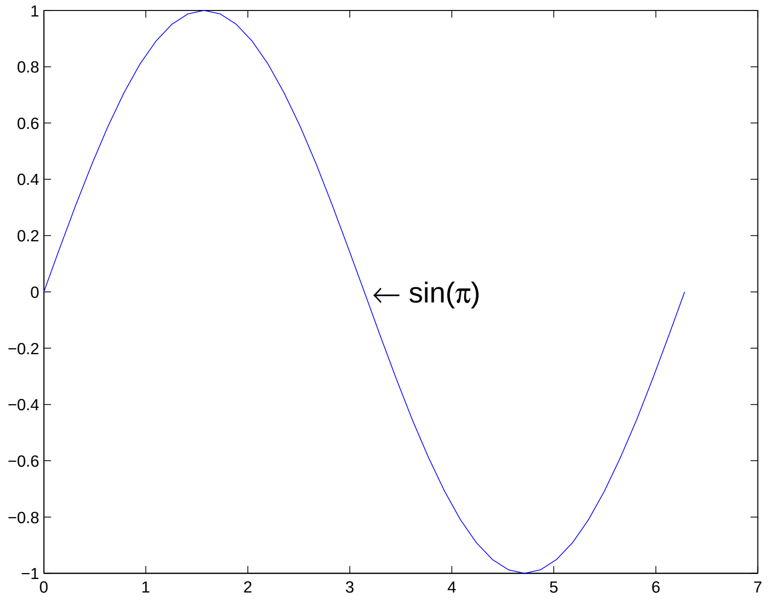
`text` does not respect the setting of the figure or axes `NextPlot` property. This allows you to add text objects to an existing axes without setting `hold` to `on`.

Examples

The statements,

```
plot(0:pi/20:2*pi,sin(0:pi/20:2*pi))
text(pi,0,' \leftarrow sin(\pi)','FontSize',18)
```

annotate the point at (`pi,0`) with the string `sin(`π`)`.



The statement,

```
text(x,y,'\ite^{i\omega\tau} = cos(\omega\tau) + i sin(\omega\tau)')
```

uses embedded TeX sequences to produce:

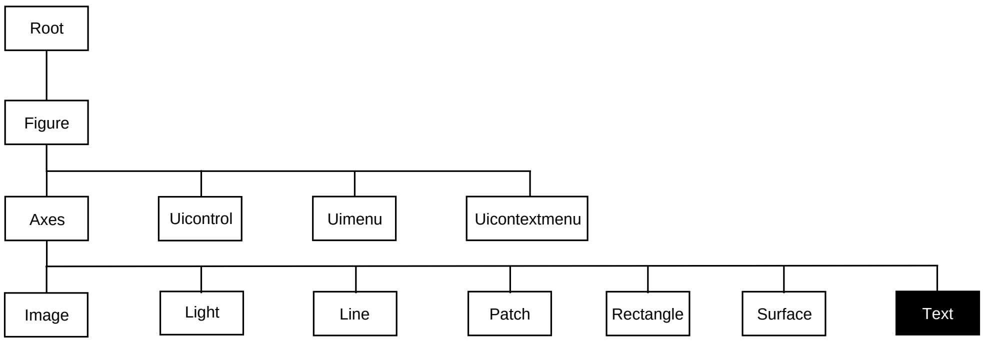
$$e^{i\omega\tau} = cos(\omega\tau) + i\ sin(\omega\tau)$$

See Also

`gtext`, `int2str`, `num2str`, `title`, `xlabel`, `ylabel`, `zlabel`

The “Labeling Graphs” topic in the online *Using MATLAB Graphics* manual discusses positioning text.

Object Hierarchy



Setting Default Properties

You can set default text properties on the axes, figure, and root levels.

```
set(0,'DefaulttextProperty',PropertyValue...)
set(gcf,'DefaulttextProperty',PropertyValue...)
set(gca,'DefaulttextProperty',PropertyValue...)
```

Where *Property* is the name of the text property and `PropertyValue` is the value you are specifying. Use `set` and `get` to access text properties.

Property List

The following table lists all text properties and provides a brief description of each. The property name links take you to an expanded description of the properties.

Property Name	Property Description	Property Value
Defining the character string		
`Editing`	Enable or disable editing mode.	Values: `on`, `off` Default: `off`
`Interpreter`	Enable or disable TeX interpretation	Values: `tex`, `none` Default: `tex`
`String`	The character string (including list of TeX character sequences)	Value: character string

Property Name	Property Description	Property Value
Positioning the character string		
Extent	Position and size of text object	Values: [left, bottom, width, height]
HorizontalAlignment	Horizontal alignment of text string	Values: left, center, right Default: left
Position	Position of text Extent rectangle	Values: [x, y, z] coordinates Default: [] empty matrix
Rotation	Orientation of text object	Values: scalar (degrees) Default: 0
Units	Units for Extent and Position properties	Values: pixels, normalized, inches, centimeters, points, data Default: data
VerticalAlignment	Vertical alignment of text string	Values: top, cap, middle, baseline, bottom Default: middle
Specifying the Font		
FontAngle	Select italic-style font	Values: normal, italic, oblique Default: normal
FontName	Select font family	Values: a font supported by your system or the string FixedWidth Default: Helvetica
FontSize	Size of font	Values: size in FontUnits Default: 10 points
FontUnits	Units for FontSize property	Values: points, normalized, inches, centimeters, pixels Default: points

Property Name	Property Description	Property Value
FontWeight	Weight of text characters	Values: light, normal, demi, bold Default: normal
Controlling the Appearance		
Clipping	Clipping to axes rectangle	Values: on, off Default: on
EraseMode	Method of drawing and erasing the text (useful for animation)	Values: normal, none, xor, background Default: normal
SelectionHighlight	Highlight text when selected (Selected property set to on)	Values: on, off Default: on
Visible	Make the text visible or invisible	Values: on, off Default: on
Color	Color of the text	ColorSpec
Controlling Access to Text Objects		
HandleVisibility	Determines if and when the the text's handle is visible to other functions	Values: on, callback, off Default: on
HitTest	Determines if the text can become the current object (see the figure CurrentObject property)	Values: on, off Default: on
General Information About Text Objects		
Children	Text objects have no children	Values: [] (empty matrix)
Parent	The parent of a text object is always an axes object	Value: axes handle
Selected	Indicate whether the text is in a "selected" state.	Values: on, off Default: off

Property Name	Property Description	Property Value
`Tag`	User-specified label	Value: any string Default: `''` (empty string)
`Type`	The type of graphics object (read only)	Value: the string `'text'`
`UserData`	User-specified data	Values: any matrix Default: `[]` (empty matrix)
Controlling Callback Routine Execution		
`BusyAction`	Specifies how to handle callback routine interruption	Values: `cancel`, `queue` Default: `queue`
`ButtonDownFcn`	Defines a callback routine that executes when a mouse button is pressed on over the text	Values: string Default: `''` (empty string)
`CreateFcn`	Defines a callback routine that executes when an text is created	Values: string Default: `''` (empty string)
`DeleteFcn`	Defines a callback routine that executes when the text is deleted (via `close` or `delete`)	Values: string Default: `''` (empty string)
`Interruptible`	Determines if callback routine can be interrupted	Values: `on`, `off` Default: `on` (can be interrupted)
`UIContextMenu`	Associates a context menu with the text	Values: handle of a uicontextmenu

Modifying Properties

You can set and query graphics object properties in two ways:

- The Property Editor is an interactive tool that enables you to see and change object property values.
- The `set` and `get` commands enable you to set and query the values of properties

To change the default value of properties see Settingcreating_plots Default Property Values.

Text Property Descriptions

This section lists property names along with the types of values each accepts. Curly braces { } enclose default values.

`BusyAction` `cancel | {queue}`

Callback routine interruption. The `BusyAction` property enables you to control how MATLAB handles events that potentially interrupt executing callback routines. If there is a callback routine executing, subsequently invoked callback routines always attempt to interrupt it. If the `Interruptible` property of the object whose callback is executing is set to `on` (the default), then interruption occurs at the next point where the event queue is processed. If the `Interruptible` property is `off`, the `BusyAction` property (of the object owning the executing callback) determines how MATLAB handles the event. The choices are:

- `cancel` – discard the event that attempted to execute a second callback routine.
- `queue` – queue the event that attempted to execute a second callback routine until the current callback finishes.

`ButtonDownFcn` string

Button press callback routine. A callback routine that executes whenever you press a mouse button while the pointer is over the text object. Define this routine as a string that is a valid MATLAB expression or the name of an M-file. The expression executes in the MATLAB workspace.

`Children` matrix (read only)

The empty matrix; text objects have no children.

Clipping on | {off}

Clipping mode. When Clipping is on, MATLAB does not display any portion of the text that is outside the axes.

Color ColorSpec

Text color. A three-element RGB vector or one of MATLAB's predefined names, specifying the text color. The default value for Color is white. See ColorSpec for more information on specifying color.

CreateFcn string

Callback routine executed during object creation. This property defines a callback routine that executes when MATLAB creates a text object. You must define this property as a default value for text. For example, the statement,

```
set(0,'DefaultTextCreateFcn',...
    'set(gcf,''Pointer'',’'crosshair'')')
```

defines a default value on the root level that sets the figure Pointer property to a crosshair whenever you create a text object. MATLAB executes this routine after setting all text properties. Setting this property on an existing text object has no effect.

The handle of the object whose CreateFcn is being executed is accessible only through the root CallbackObject property, which you can query using gcbo.

DeleteFcn string

Delete text callback routine. A callback routine that executes when you delete the text object (e.g., when you issue a delete command or clear the axes or figure). MATLAB executes the routine before destroying the object's properties so these values are available to the callback routine.

The handle of the object whose DeleteFcn is being executed is accessible only through the root CallbackObject property, which you can query using gcbo.

Editing on | {off}

Enable or disable editing mode. When this property is set to the default off, you cannot edit the text string interactively (i.e., you must change the String property to change the text). When this property is set to on, MATLAB places an insert cursor at the beginning of the text string and enables editing. To apply the new text string:

- Press the **ESC** key
- Clicking in any figure window (including the current figure)
- Reset the `Editing` property to `off`

MATLAB then updates the `String` property to contain the new text and resets the `Editing` property to `off`. You must reset the `Editing` property to on to again resume editing.

EraseMode `{normal} | none | xor | background`

Erase mode. This property controls the technique MATLAB uses to draw and erase text objects. Alternative erase modes are useful for creating animated sequences, where controlling the way individual object redraw is necessary to improve performance and obtain the desired effect.

- `normal` — Redraw the affected region of the display, performing the three-dimensional analysis necessary to ensure that all objects are rendered correctly. This mode produces the most accurate picture, but is the slowest. The other modes are faster, but do not perform a complete redraw and are therefore less accurate.
- `none` — Do not erase the text when it is moved or destroyed. While the object is still visible on the screen after erasing with `EraseMode none`, you cannot print it because MATLAB stores no information about its former location.
- `xor` — Draw and erase the text by performing an exclusive OR (XOR) with each pixel index of the screen beneath it. When the text is erased, it does not damage the objects beneath it. However, when text is drawn in `xor` mode, its color depends on the color of the screen beneath it and is correctly colored only when over axes background `Color`, or the figure background `Color` if the axes `Color` is set to `none`.
- `background` — Erase the text by drawing it in the background `Color`, or the figure background `Color` if the axes `Color` is set to `none`. This damages objects that are behind the erased text, but text is always properly colored.

Printing with Non-normal Erase Modes. MATLAB always prints figures as if the `EraseMode` of all objects is `normal`. This means graphics objects created with `EraseMode` set to `none`, `xor`, or `background` can look different on screen than on paper. On screen, MATLAB may mathematically combine layers of colors (e.g., XORing a pixel color with that of the pixel behind it) and ignore three-dimensional sorting to obtain greater rendering speed. However, these techniques are not applied to the printed output.

You can use the MATLAB `getframe` command or other screen capture application to create an image of a figure containing non-normal mode objects.

`Extent` position rectangle (read only)

Position and size of text. A four-element read-only vector that defines the size and position of the text string.

```
[left,bottom,width,height]
```

If the `Units` property is set to `data` (the default), `left` and `bottom` are the *x* and *y* coordinates of the lower-left corner of the text `Extent` rectangle.

For all other values of `Units`, `left` and `bottom` are the distance from the lower-left corner of the axes position rectangle to the lower-left corner of the text `Extent` rectangle. `width` and `height` are the dimensions of the `Extent` rectangle. All measurements are in units specified by the `Units` property.

`FontAngle` `{normal} | italic | oblique`

Character slant. MATLAB uses this property to select a font from those available on your particular system. Generally, setting this property to `italic` or `oblique` selects a slanted font.

`FontName` A name such as `Courier` or the string `FixedWidth`

Font family. A string specifying the name of the font to use for the text object. To display and print properly, this must be a font that your system supports. The default font is Helvetica.

Specifying a Fixed-Width Font

If you want text to use a fixed-width font that looks good in any locale, you should set `FontName` to the string `FixedWidth`:

```
set(text_handle,'FontName','FixedWidth')
```

This eliminates the need to hardcode the name of a fixed-width font, which may not display text properly on systems that do not use ASCII character encoding (such as in Japan where multibyte character sets are used). A properly written MATLAB application that needs to use a fixed-width font should set `FontName` to `FixedWidth` (note that this string is case sensitive) and rely on `FixedWidthFontName` to be set correctly in the end-user's environment.

End users can adapt a MATLAB application to different locales or personal environments by setting the root `FixedWidthFontName` property to the appropriate value for that locale from `startup.m`.

Note that setting the root `FixedWidthFontName` property causes an immediate update of the display to use the new font.

`FontSize` size in `FontUnits`

Font size. An integer specifying the font size to use for text, in units determined by the `FontUnits` property. The default point size is 10 (1 point = 1/72 inch).

`FontWeight` `light | {normal} | demi | bold`

Weight of text characters. MATLAB uses this property to select a font from those available on your particular system. Generally, setting this property to `bold` or `demi` causes MATLAB to use a bold font.

`FontUnits` `{points} | normalized | inches | centimeters | pixels`

Font size units. MATLAB uses this property to determine the units used by the `FontSize` property. `Normalized` units interpret `FontSize` as a fraction of the height of the parent axes. When you resize the axes, MATLAB modifies the screen `FontSize` accordingly. `pixels`, `inches`, `centimeters`, and `points` are absolute units (1 point = 1/72 inch).

`HandleVisibility` `{on} | callback | off`

Control access to object's handle by command-line users and GUIs. This property determines when an object's handle is visible in its parent's list of children. `HandleVisibility` is useful for preventing command-line users from accidentally drawing into or deleting a figure that contains only user interface devices (such as a dialog box).

Handles are always visible when `HandleVisibility` is `on`.

Setting `HandleVisibility` to `callback` causes handles to be visible from within callback routines or functions invoked by callback routines, but not from within functions invoked from the command line. This provides a means to protect GUIs from command-line users, while allowing callback routines to have complete access to object handles.

Setting `HandleVisibility` to `off` makes handles invisible at all times. This may be necessary when a callback routine invokes a function that might potentially damage the GUI (such as evaluating a user-typed string), and so temporarily hides its own handles during the execution of that function.

When a handle is not visible in its parent's list of children, it cannot be returned by functions that obtain handles by searching the object hierarchy or querying handle properties. This includes `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close`.

When a handle's visibility is restricted using `callback` or `off`, the object's handle does not appear in its parent's `Children` property, figures do not appear in the root's `CurrentFigure` property, objects do not appear in the root's `CallbackObject` property or in the figure's `CurrentObject` property, and axes do not appear in their parent's `CurrentAxes` property.

You can set the root `ShowHiddenHandles` property to `on` to make all handles visible, regardless of their `HandleVisibility` settings (this does not affect the values of the `HandleVisibility` properties).

Handles that are hidden are still valid. If you know an object's handle, you can `set` and `get` its properties, and pass it to any function that operates on handles.

HitTest `{on} | off`

Selectable by mouse click. `HitTest` determines if the text can become the current object (as returned by the `gco` command and the figure `CurrentObject` property) as a result of a mouse click on the text. If `HitTest` is `off`, clicking on the text selects the object below it (which is usually the axes containing it).

For example, suppose you define the button down function of an image (see the `ButtonDownFcn` property) to display text at the location you click on with the mouse.

First define the callback routine.

```
function bd_function
pt = get(gca,'CurrentPoint');
text(pt(1,1),pt(1,2),pt(1,3),...
   '{\fontsize{20}\oplus} The spot to label',...
   'HitTest','off')
```

Now display an image, setting its `ButtonDownFcn` property to the callback routine.

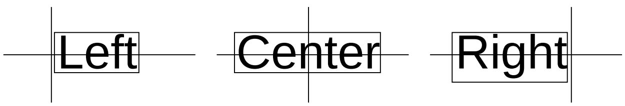
```
load earth
image(X,'ButtonDownFcn','bd_function'); colormap(map)
```

When you click on the image, MATLAB displays the text string at that location. With `HitTest` set to `off`, existing text cannot intercept any subsequent button down events that occur over the text. This enables the image's button down function to execute.

`HorizontalAlignment` `{left} | center | right`

Horizontal alignment of text. This property specifies the horizontal justification of the text string. It determines where MATLAB places the string with regard to the point specified by the `Position` property. The following picture illustrates the alignment options.

Text `HorizontalAlignment` property viewed with the `VerticalAlignment` property set to `middle` (the default).



See the `Extent` property for related information.

`Interpreter` `{tex} | none`

Interpret Tex instructions. This property controls whether MATLAB interprets certain characters in the `String` property as Tex instructions (default) or displays all characters literally. See the `String` property for a list of support Tex instructions.

`Interruptible` `{on} | off`

Callback routine interruption mode. The `Interruptible` property controls whether a text callback routine can be interrupted by subsequently invoked callback routines. text objects have four properties that define callback routines: `ButtonDownFcn`, `CreateFcn`, and `DeleteFcn`. See the `BusyAction` property for information on how MATLAB executes callback routines.

`Parent` handle

Text object's parent. The handle of the text object's parent object. The parent of a text object is the axes in which it is displayed. You can move a text object to another axes by setting this property to the handle of the new parent.

`Position` `[x,y,[z]]`

Location of text. A two- or three-element vector, `[x y [z]]`, that specifies the location of the text in three dimensions. If you omit the `z` value, it defaults to 0. All measurements are in units specified by the `Units` property. Initial value is `[0 0 0]`.

`Rotation` scalar (default = 0)

Text orientation. This property determines the orientation of the text string. Specify values of rotation in degrees (positive angles cause counterclockwise rotation).

`Selected` `on | {off}`

Is object selected? When this property is `on`, MATLAB displays selection handles if the `SelectionHighlight` property is also `on`. You can, for example, define the `ButtonDownFcn` to set this property, allowing users to select the object with the mouse.

`SelectionHighlight` `{on} | off`

Objects highlight when selected. When the `Selected` property is `on`, MATLAB indicates the selected state by drawing four edge handles and four corner handles. When `SelectionHighlight` is `off`, MATLAB does not draw the handles.

`String` string

The text string. Specify this property as a quoted string for single-line strings, or as a cell array of strings or a padded string matrix for multiline strings. MATLAB displays this string at the specified location. Vertical slash characters are not interpreted as linebreaks in text strings, and are drawn as part of the text string. See the "Remarks" section for more information.

When the text `Interpreter` property is `Tex` (the default), you can use a subset of TeX commands embedded in the string to produce special characters such as

Greek letters and mathematical symbols. The following table lists these characters and the character sequence used to define them.

Character Sequence	Symbol	Character Sequence	Symbol	Character Sequence	Symbol
\alpha	α	\upsilon	υ	\sim	~
\beta	β	\phi	φ	\leq	≤
\gamma	γ	\chi	χ	\infty	∞
\delta	δ	\psi	ψ	\clubsuit	♣
\epsilon	ε	\omega	ω	\diamondsuit	♦
\zeta	ζ	\Gamma	Γ	\heartsuit	♥
\eta	η	\Delta	Δ	\spadesuit	♠
\theta	θ	\Theta	Θ	\leftrightarrow	↔
\vartheta	ϑ	\Lambda	Λ	\leftarrow	←
\iota	ι	\Xi	Ξ	\uparrow	↑
\kappa	κ	\Pi	Π	\rightarrow	→
\lambda	λ	\Sigma	Σ	\downarrow	↓
\mu	µ	\Upsilon	ϒ	\circ	°
\nu	ν	\Phi	Φ	\pm	±
\xi	ξ	\Psi	Ψ	\geq	≥
\pi	π	\Omega	Ω	\propto	∝
\rho	ρ	\forall	∀	\partial	∂
\sigma	σ	\exists	∃	\bullet	•
\varsigma	ς	\ni	∍	\div	÷
\tau	τ	\cong	≅	\neq	≠

Character Sequence	Symbol	Character Sequence	Symbol	Character Sequence	Symbol
\equiv	≡	\approx	≈	\aleph	ℵ
\Im	ℑ	\Re	ℜ	\wp	℘
\otimes	⊗	\oplus	⊕	\oslash	∅
\cap	∩	\cup	∪	\supseteq	⊇
\supset	⊃	\subseteq	⊆	\subset	⊂
\int	∫	\in	∈	\o	ο
\rfloor	⌋	\lceil	⌈	\nabla	∇
\lfloor	⌊	\cdot	·	\ldots	...
\perp	⊥	\neg	¬	\prime	′
\wedge	∧	\times	×	\0	∅
\rceil	⌉	\surd	√	\mid	|
\vee	∨	\varpi	ϖ	\copyright	©
\langle	〈	\rangle	〉		

You can also specify stream modifiers that control the font used. The first four modifiers are mutually exclusive. However, you can use `\fontname` in combination with one of the other modifiers:

- `\bf` – bold font
- `\it` – italics font
- `\sl` – oblique font (rarely available)
- `\rm` – normal font
- `\fontname{`*`fontname`*`}` – specify the name of the font family to use.
- `\fontsize{`*`fontsize`*`}` – specify the font size in `FontUnits`.

Stream modifiers remain in effect until the end of the string or only within the context defined by braces { }.

Specifying Subscript and Superscript Characters

The subscript character "_" and the superscript character "^" modify the character or substring defined in braces immediately following.

To print the special characters used to define the Tex strings when `Interpreter` is `Tex`, prefix them with the backslash "\" character: \\, \{, \} _, \^.

See the example for more information.

When `Interpreter` is `none`, no characters in the `String` are interpreted, and all are displayed when the text is drawn.

`Tag` string

User-specified object label. The `Tag` property provides a means to identify graphics objects with a user-specified label. This is particularly useful when constructing interactive graphics programs that would otherwise need to define object handles as global variables or pass them as arguments between callback routines. You can define `Tag` as any string.

`Type` string (read only)

Class of graphics object. For text objects, `Type` is always the string `'text'`.

`Units` `pixels | normalized | inches | centimeters | points | {data}`

Units of measurement. This property specifies the units MATLAB uses to interpret the `Extent` and `Position` properties. All units are measured from the lower-left corner of the axes plotbox. `Normalized` units map the lower-left corner of the rectangle defined by the axes to (0,0) and the upper-right corner to (1.0,1.0). `pixels`, `inches`, `centimeters`, and `points` are absolute units (1 point = $^1/_{72}$ inch). `data` refers to the data units of the parent axes.

If you change the value of `Units`, it is good practice to return it to its default value after completing your computation so as not to affect other functions that assume `Units` is set to the default value.

`UserData` matrix

User-specified data. Any data you want to associate with the text object. MATLAB does not use this data, but you can access it using `set` and `get`.

UIContextMenu handle of a uicontextmenu object

Associate a context menu with the text. Assign this property the handle of a uicontextmenu object created in the same figure as the text. Use the `uicontextmenu` function to create the context menu. MATLAB displays the context menu whenever you right-click over the text.

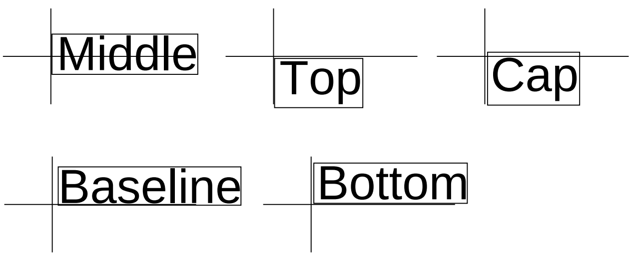
VerticalAlignment `top | cap | {middle} | baseline | bottom`

Vertical alignment of text. This property specifies the vertical justification of the text string. It determines where MATLAB places the string with regard to the value of the `Position` property. The possible values mean:

- `top` – Place the top of the string's `Extent` rectangle at the specified *y*-position.
- `cap` – Place the string so that the top of a capital letter is at the specified *y*-position.
- `middle` – Place the middle of the string at specified *y*-position.
- `baseline` – Place font baseline at the specified *y*-position.
- `bottom` – Place the bottom of the string's `Extent` rectangle at the specified *y*-position.

The following picture illustrates the alignment options.

Text `VerticalAlignment` property viewed with the `HorizontalAlignment` property set to `left` (the default).



Visible `{on} | off`

Text visibility. By default, all text is visible. When set to `off`, the text is not visible, but still exists and you can query and set its properties.

Purpose

Read formatted data from text file

Graphical Interface

As an alternative to `textread`, use the Import Wizard. To activate the Import Wizard, select **Import Data** from the **File** menu.

Syntax

```
[A,B,C,...] = textread('filename','format')
[A,B,C,...] = textread('filename','format',N)
[...] = textread(...,'param','value',...)
```

Description

`[A,B,C,...] = textread('filename','format')` reads data from the file `'filename'` into the variables `A,B,C,` and so on, using the specified `format`, until the entire file is read. `textread` is useful for reading text files with a known format. Both fixed and free format files can be handled.

`textread` matches and converts groups of characters from the input. Each input field is defined as a string of non-whitespace characters that extends to the next whitespace or delimiter character, or to the maximum field width. Repeated delimiter characters are significant, while repeated whitespace characters are treated as one.

The `format` string determines the number and types of return arguments. The number of return arguments is the number of items in the `format` string. The `format` string supports a subset of the conversion specifiers and conventions of the C language `fscanf` routine. Values for the `format` string are listed in the table below. Whitespace characters in the `format` string are ignored.

format	Action	Output
Literals (ordinary characters)	Ignore the matching characters. For example, in a file that has `Dept` followed by a number (for department number), to skip the `Dept` and read only the number, use `'Dept'` in the `format` string.	None
`%d`	Read a signed integer value.	Double array
`%u`	Read an integer value.	Double array
`%f`	Read a floating point value.	Double array

format	Action	Output
`%s`	Read a whitespace or delimiter-separated string.	Cell array of strings
`%q`	Read a string, which could be in double quotes.	Cell array of strings. Does not include the double quotes.
`%c`	Read characters, including white space.	Character array
`%[...]`	Read the longest string containing characters specified in the brackets.	Cell array of strings
`%[^...]`	Read the longest non-empty string containing characters that are not specified in the brackets.	Cell array of strings
`%*...` instead of `%`	Ignore the matching characters specified by `*`.	No output
`%w...` instead of `%`	Read field width specified by `w`. The `%f` format supports `%w.pf`, where `w` is the field width and `p` is the precision.	

`[A,B,C,...] = textread('filename','format',N)` reads the data, reusing the `format` string `N` times, where `N` is an integer greater than zero. If `N` is smaller than zero, `textread` reads the entire file.

[...] = textread(...,'param','value',...) customizes textread using param/value pairs, as listed in the table below.

param	value	Action
whitespace	Any from the list below:	Treats vector of characters as whitespace. Default is ' \b\t'.
	' ' \b \n \r \t	Space Backspace New line Carriage return Horizontal tab
delimiter	Delimiter character	Specifies delimiter character. Default is none.
expchars	Exponent characters	Default is eEdD.
bufsize	positive integer	Specifies the maximum string length, in bytes. Default is 4095.
headerlines	positive integer	Ignores the specified number of lines at the beginning of the file.
commentstyle	matlab	Ignores characters after %
commentstyle	shell	Ignores characters after #.
commentstyle	c	Ignores characters between /* and */.
commentstyle	c++	Ignores characters after //.

Note When textread reads a consecutive series of whitespace values, it treats them as one whitespace. When it reads a consecutive series of delimiter values, it treats each as a separate delimiter.

Examples

Example 1 – Read All Fields in Free Format File Using %

The first line of `mydata.dat` is

```
Sally    Type1 12.34 45 Yes
```

Read the first line of the file as a free format file using the `%` format.

```
[names,types,x,y,answer] = textread('mydata.dat','%s %s %f ...
   %d %s',1)
```

returns

```
names =
    'Sally'
types =
    'Type1'
x =
  12.34000000000000
y =
    45
answer =
    'Yes'
```

Example 2 – Read as Fixed Format File, Ignoring the Floating Point Value

The first line of `mydata.dat` is

```
Sally    Type1 12.34 45 Yes
```

Read the first line of the file as a fixed format file, ignoring the floating point value.

```
[names,types,y,answer] = textread('mydata.dat','%9c %5s %*f ...
%2d %3s',1)
```

returns

```
names =
Sally
types =
    'Type1'
y =
    45
answer =
```

```
    'Yes'
```

`%*f` in the `format` string causes `textread` to ignore the floating point value, in this case, `12.34`.

Example 3 – Read Using Literal to Ignore Matching Characters

The first line of `mydata.dat` is

```
Sally    Type1 12.34 45 Yes
```

Read the first line of the file, ignoring the characters `Type` in the second field.

```
[names,typenum,x,y,answer] = textread('mydata.dat','%s Type%d %f
%d %s',1)
```

returns

```
names =
    'Sally'
typenum =
     1
x =
  12.34000000000000
y =
    45
answer =
    'Yes'
```

`Type%d` in the `format` string causes the characters `Type` in the second field to be ignored, while the rest of the second field is read as a signed integer, in this case, `1`.

Example 4 – Read M-file into a Cell Array of Strings

Read the file `fft.m` into cell array of strings.

```
file = textread('fft.m','%s','delimiter','\n','whitespace','');
```

See Also

`dlmread`, `csvread`, `fscanf`

textwrap

Purpose

Return wrapped string matrix for given uicontrol

Syntax

```
outstring = textwrap(h,instring)
[outstring,position] = textwrap(h,instring)
```

Description

`outstring = textwrap(h,instring)` returns a wrapped string cell array, `outstring`, that fits inside the uicontrol with handle `h`. `instring` is a cell array, with each cell containing a single line of text. `outstring` is the wrapped string matrix in cell array format. Each cell of the input string is considered a paragraph.

`[outstring,position]=textwrap(h,instring)` returns the recommended position of the uicontrol in the units of the uicontrol. `position` considers the extent of the multiline text in the *x* and *y* directions.

Example

Place a textwrapped string in a uicontrol:

```
pos = [10 10 100 10];
h = uicontrol('Style','Text','Position',pos);
string = {'This is a string for the uicontrol.',
          'It should be correctly wrapped inside.'};
[outstring,newpos] = textwrap(h,string);
pos(4) = newpos(4);
set(h,'String',outstring,'Position',[pos(1),pos(2),pos(3)+10,pos(4)])
```

See Also

`uicontrol`

Purpose Stopwatch timer

Syntax

```
tic
   any statements
toc
t = toc
```

Description `tic` starts a stopwatch timer.

`toc` prints the elapsed time since `tic` was used.

`t = toc` returns the elapsed time in `t`.

Examples This example measures how the time required to solve a linear system varies with the order of a matrix.

```
for n = 1:100
    A = rand(n,n);
    b = rand(n,1);
    tic
    x = A\b;
    t(n) = toc;
end
plot(t)
```

See Also `clock`, `cputime`, `etime`

title

Purpose

Add title to current axes

Syntax

```
title('string')
title(fname)
title(...,'PropertyName',PropertyValue,...)
h = title(...)
```

Description

Each axes graphics object can have one title. The title is located at the top and in the center of the axes.

`title('string')` outputs the string at the top and in the center of the current axes.

`title(fname)` evaluates the function that returns a string and displays the string at the top and in the center of the current axes.

`title(...,'`*`PropertyName`*`',PropertyValue,...)` specifies property name and property value pairs for the text graphics object that `title` creates.

`h = title(...)` returns the handle to the text object used as the title.

Examples

Display today's date in the current axes:

```
title(date)
```

Include a variable's value in a title:

```
f = 70;
c = (f–32)/1.8;
title(['Temperature is ',num2str(c),'C'])
```

Include a variable's value in a title and set the color of the title to yellow:

```
n = 3;
title(['Case number #',int2str(n)],'Color','y')
```

Include Greek symbols in a title:

```
title('\ite^{\omega\tau} = cos(\omega\tau) + isin(\omega\tau)’)
```

Include a superscript character in a title:

```
title('\alpha^2’)
```

Include a subscript character in a title:

```
title('X_1')
```

The text object `String` property lists the available symbols.

Remarks

`title` sets the `Title` property of the current axes graphics object to a new text graphics object. See the text `String` property for more information.

See Also

`gtext`, `int2str`, `num2str`, `plot`, `text`, `xlabel`, `ylabel`, `zlabel`

toeplitz

Purpose

Toeplitz matrix

Syntax

```
T = toeplitz(c,r)
T = toeplitz(r)
```

Description

A *Toeplitz* matrix is defined by one row and one column. A *symmetric Toeplitz* matrix is defined by just one row. `toeplitz` generates Toeplitz matrices given just the row or row and column description.

`T = toeplitz(c,r)` returns a nonsymmetric Toeplitz matrix `T` having `c` as its first column and `r` as its first row. If the first elements of `c` and `r` are different, a message is printed and the column element is used.

`T = toeplitz(r)` returns the symmetric or Hermitian Toeplitz matrix formed from vector `r`, where `r` defines the first row of the matrix.

Examples

A Toeplitz matrix with diagonal disagreement is

```
c = [1  2  3  4  5];
r = [1.5  2.5  3.5  4.5  5.5];
toeplitz(c,r)
Column wins diagonal conflict:
ans =
    1.000    2.500    3.500    4.500    5.500
    2.000    1.000    2.500    3.500    4.500
    3.000    2.000    1.000    2.500    3.500
    4.000    3.000    2.000    1.000    2.500
    5.000    4.000    3.000    2.000    1.000
```

See Also

`hankel`

Purpose Sum of diagonal elements

Syntax

```
b = trace(A)
```

Description `b = trace(A)` is the sum of the diagonal elements of the matrix `A`.

Algorithm `trace` is a single-statement M-file.

```
t = sum(diag(A));
```

See Also `det`, `eig`

trapz

Purpose

Trapezoidal numerical integration

Syntax

```
Z = trapz(Y)
Z = trapz(X,Y)
Z = trapz(...,dim)
```

Description

`Z = trapz(Y)` computes an approximation of the integral of `Y` via the trapezoidal method (with unit spacing). To compute the integral for spacing other than one, multiply `Z` by the spacing increment.

If `Y` is a vector, `trapz(Y)` is the integral of `Y`.

If `Y` is a matrix,`trapz(Y)` is a row vector with the integral over each column.

If `Y` is a multidimensional array, `trapz(Y)` works across the first nonsingleton dimension.

`Z = trapz(X,Y)` computes the integral of `Y` with respect to `X` using trapezoidal integration.

If `X` is a column vector and `Y` an array whose first nonsingleton dimension is `length(X)`, `trapz(X,Y)` operates across this dimension.

`Z = trapz(...,dim)` integrates across the dimension of `Y` specified by scalar `dim`. The length of `X`, if given, must be the same as `size(Y,dim)`.

Examples

The exact value of $\int_0^{\pi} \sin(x)\,dx$ is 2.

To approximate this numerically on a uniformly spaced grid, use

```
X = 0:pi/100:pi;
Y = sin(x);
```

Then both

```
Z = trapz(X,Y)
```

and

```
Z = pi/100*trapz(Y)
```

produce

```
Z =
    1.9998
```

A nonuniformly spaced example is generated by

```
X = sort(rand(1,101)*pi);
Y = sin(X);
Z = trapz(X,Y);
```

The result is not as accurate as the uniformly spaced grid. One random sample produced

```
Z =
    1.9984
```

See Also

`cumsum`, `cumtrapz`

treelayout

Purpose Lay out tree or forest

Syntax

```
[x,y] = treelayout(parent,post)
[x,y,h,s] = treelayout(parent,post)
```

Description

`[x,y] = treelayout(parent,post)` lays out a tree or a forest. `parent` is the vector of parent pointers, with `0` for a root. `post` is an optional postorder permutation on the tree nodes. If you omit `post`, `treelayout` computes it. `x` and `y` are vectors of coordinates in the unit square at which to lay out the nodes of the tree to make a nice picture.

`[x,y,h,s] = treelayout(parent,post)` also returns the height of the tree `h` and the number of vertices `s` in the top-level separator.

See Also `etree`, `treeplot`, `etreeplot`, `symbfact`

Purpose Plot picture of tree

Syntax

```
treeplot(p)
treeplot(p,nodeSpec,edgeSpec)
```

Description `treeplot(p)` plots a picture of a tree given a vector of parent pointers, with `p(i)` = 0 for a root.

`treeplot(p,nodeSpec,edgeSpec)` allows optional parameters `nodeSpec` and `edgeSpec` to set the node or edge color, marker, and linestyle. Use `''` to omit one or both.

See Also `etree`, `etreeplot`, `treelayout`

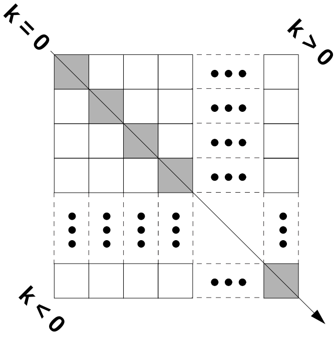
Purpose Lower triangular part of a matrix

Syntax

```
L = tril(X)
L = tril(X,k)
```

Description `L = tril(X)` returns the lower triangular part of `X`.

`L = tril(X,k)` returns the elements on and below the kth diagonal of `X`. `k = 0` is the main diagonal, `k > 0` is above the main diagonal, and `k < 0` is below the main diagonal.



Examples

```
tril(ones(4,4),-1)

ans =

     0     0     0     0
     1     0     0     0
     1     1     0     0
     1     1     1     0
```

See Also `diag`, `triu`

Purpose Triangular mesh plot

Syntax

```
trimesh(Tri,X,Y,Z)
trimesh(Tri,X,Y,Z,C)
trimesh(...'PropertyName',PropertyValue...)
h = trimesh(...)
```

Description

`trimesh(Tri,X,Y,Z)` displays triangles defined in the *m*-by-3 face matrix `Tri` as a mesh. Each row of `Tri` defines a single triangular face by indexing into the vectors or matrices that contain the `X`, `Y`, and `Z` vertices.

`trimesh(Tri,X,Y,Z,C)` specifies color defined by `C` in the same manner as the `surf` function. MATLAB performs a linear transformation on this data to obtain colors from the current colormap.

`trimesh(...'`*`PropertyName`*`',PropertyValue...)` specifies additional patch property names and values for the patch graphics object created by the function.

`h = trimesh(...)` returns a handle to a patch graphics object.

Example

Create vertex vectors and a face matrix, then create a triangular mesh plot.

```
x = rand(1,50);
y = rand(1,50);
z = peaks(6*x–3,6*x–3);
tri = delaunay(x,y);
trimesh(tri,x,y,z)
```

See Also

`patch`, `tetramesh`, `triplot`, `trisurf`, `delaunay`

Purpose 2-D triangular plot

Syntax

```
triplot(TRI,x,y)
triplot(TRI,x,y,color)
h = triplot(...)
triplot(...,'param','value','param','value'...)
```

Description

`triplot(TRI,x,y)` displays the triangles defined in the `m`-by-3 matrix `TRI`. A row of `TRI` contains indices into the vectors `x` and `y` that define a single triangle. The default line color is blue.

`triplot(TRI,x,y,color)` uses the string `color` as the line color. `color` can also be a line specification. See `ColorSpec` for a list of valid color strings. See `LineSpec` for information about line specifications.

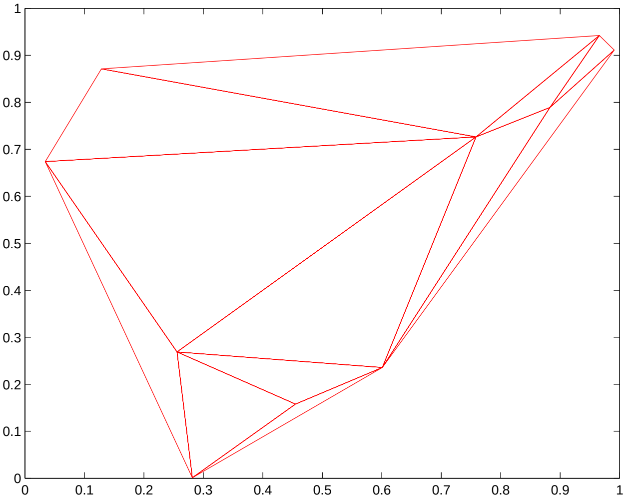
`h = triplot(...)` returns a vector of handles to the displayed triangles.

`triplot(...,'param','value','param','value'...)` allows additional line property name/property value pairs to be used when creating the plot. See `Line Properties` for information about the available properties.

Examples

This code plots the Delaunay triangulation for 10 randomly generated points.

```
rand('state',7);
x = rand(1,10);
y = rand(1,10);
TRI = delaunay(x,y);
triplot(TRI,x,y,'red')
```



See Also

`ColorSpec`, `delaunay`, `line`, `Line Properties`, `LineSpec`, `plot`, `trimesh`, `trisurf`

trisurf

Purpose Triangular surface plot

Syntax

```
trisurf(Tri,X,Y,Z)
trisurf(Tri,X,Y,Z,C)
trisurf(...'PropertyName',PropertyValue...)
h = trisurf(...)
```

Description

`trisurf(Tri,X,Y,Z)` displays triangles defined in the *m*-by-3 face matrix `Tri` as a surface. Each row of `Tri` defines a single triangular face by indexing into the vectors or matrices that contain the `X`, `Y`, and `Z` vertices.

`trisurf(Tri,X,Y,Z,C)` specifies color defined by `C` in the same manner as the `surf` function. MATLAB performs a linear transformation on this data to obtain colors from the current colormap.

`trisurf(...'`*`PropertyName`*`',PropertyValue...)` specifies additional patch property names and values for the patch graphics object created by the function.

`h = trisurf(...)` returns a patch handle.

Example Create vertex vectors and a face matrix, then create a triangular surface plot.

```
x = rand(1,50);
y = rand(1,50);
z = peaks(6*x–3,6*x–3);
tri = delaunay(x,y);
trisurf(tri,x,y,z)
```

See Also `patch`, `surf`, `tetramesh`, `trimesh`, `triplot`, `delaunay`

Purpose

Upper triangular part of a matrix

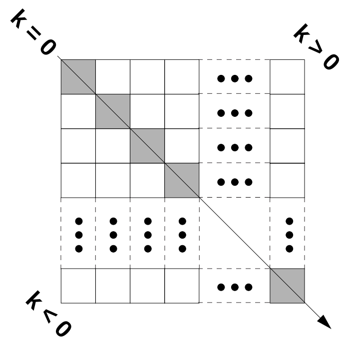
Syntax

```
U = triu(X)
U = triu(X,k)
```

Description

`U = triu(X)` returns the upper triangular part of `X`.

`U = triu(X,k)` returns the element on and above the kth diagonal of `X`. `k = 0` is the main diagonal, `k > 0` is above the main diagonal, and `k < 0` is below the main diagonal.



Examples

```
triu(ones(4,4),-1)

ans =

    1    1    1    1
    1    1    1    1
    0    1    1    1
    0    0    1    1
```

See Also

`diag`, `tril`

try

Purpose Begin `try` block

Description The general form of a `try` statement is:

```
try,
   statement,
   ...,
   statement,
catch,
   statement,
   ...,
   statement,
end
```

Normally, only the statements between the `try` and `catch` are executed. However, if an error occurs while executing any of the statements, the error is captured into `lasterr`, and the statements between the `catch` and `end` are executed. If an error occurs within the `catch` statements, execution stops unless caught by another `try...catch` block. The error string produced by a failed `try` block can be obtained with `lasterr`.

See Also `catch`, `end`, `eval`, `evalin`

Purpose Search for enclosing Delaunay triangle

Syntax

```
T = tsearch(x,y,TRI,xi,yi)
```

Description

`T = tsearch(x,y,TRI,xi,yi)` returns an index into the rows of `TRI` for each point in `xi`, `yi`. The `tsearch` command returns `NaN` for all points outside the convex hull. Requires a triangulation `TRI` of the points `x,y` obtained from `delaunay`.

Note `tsearch` is based on `qhull` [2]. For information about `qhull`, see `http://www.geom.umn.edu/software/qhull/`. For copyright information, see `http://www.geom.umn.edu/software/download/COPYING.html`.

See Also `delaunay`, `delaunayn`, `dsearch`, `tsearchn`

References

[1] Barber, C. B., D.P. Dobkin, and H.T. Huhdanpaa, "The Quickhull Algorithm for Convex Hulls," *ACM Transactions on Mathematical Software*, Vol. 22, No. 4, Dec. 1996, p. 469-483. Available in HTML format at `http://www.acm.org/pubs/citations/journals/toms/1996-22-4/p469-barber/` and in PostScript format at `ftp://geom.umn.edu/pub/software/qhull-96.ps`.

[2] National Science and Technology Research Center for Computation and Visualization of Geometric Structures (The Geometry Center), University of Minnesota. 1993.

tsearchn

Purpose n-D closest simplex search

Syntax

```
t = tsearchn(X,TES,XI)
[t,P] = tsearchn(X,TES,XI)
```

Description `t = tsearchn(X,TES,XI)` returns the indices `t` of the enclosing simplex of the Delaunay tessellation `TES` for each point in `XI`. `X` is an `m`-by-`n` matrix, representing `m` points in `n`-D space. `XI` is a `p`-by-`n` matrix, representing `p` points in `n`-D space. `tsearchn` returns `NaN` for all points outside the convex hull of `X`. `tsearchn` requires a tessellation `TES` of the points `X` obtained from `delaunayn`.

`[t,P] = tsearchn(X,TES,XI)` also returns the Barencentric coordinate `P` of `XI` in the simplex `TES`. `P` is a `p`-by-`n+1` matrix. Each row of `P` is the Barencentric coordinate of the corresponding point in `XI`. It is useful for interpolation.

See Also `delaunayn`, `griddatan`, `tsearch`

Purpose List file

Syntax

```
type ('filename')
type filename
```

Description `type('filename')` displays the contents of the specified file in the MATLAB Command Window. Use the full path for `filename`, or use a MATLAB relative partial pathname.

If you do not specify a filename extension there is no `filename` file without an extension, the `type` function adds the `.m` extension by default. The `type` function checks the directories specified in MATLAB's search path, which makes it convenient for listing the contents of M-files on the screen.

`type filename` is the unquoted form of the syntax.

Examples `type('foo.bar')` lists the contents of the file `foo.bar`.

`type foo` lists the contents of the file `foo`. If `foo` does not exist, `type foo` lists the contents of the file `foo.m`.

See Also `cd`, `dbtype`, `delete`, `dir`, `partialpath`, `path`, `what`, `who`

uicontextmenu

Purpose Create a context menu

Syntax

```
handle = uicontextmenu('PropertyName',PropertyValue,...);
```

Description uicontextmenu creates a context menu, which is a menu that appears when the user right-clicks on a graphics object.

You create context menu items using the uimenu function. Menu items appear in the order the uimenu statements appear. You associate a context menu with an object using the UIContextMenu property for the object and specifying the context menu's handle as the property value.

Properties This table lists the properties that are useful to uicontextmenu objects, grouping them by function. Each property name acts as a link to a description of the property.

Property Name	Property Description	Property Value
Controlling Style and Appearance		
Visible	Uicontextmenu visibility	Value: on, off Default: off
Position	Location of uicontextmenu when Visible is set to on	Value: two-element vector Default: [0 0]
General Information About the Object		
Children	The uimenus defined for the uicontextmenu	Value: matrix
Parent	Uicontextmenu object's parent	Value: scalar figure handle
Tag	User-specified object identifier	Value: string
Type	Class of graphics object	Value: string (read-only) Default: uicontrol
UserData	User-specified data	Value: matrix

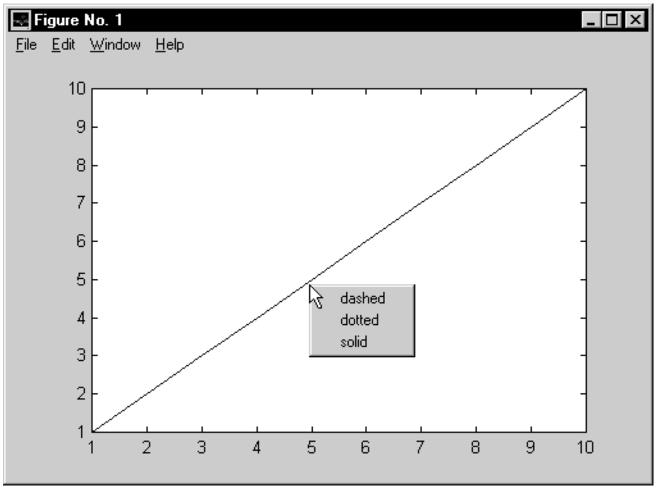
Property Name	Property Description	Property Value
Controlling Callback Routine Execution		
`BusyAction`	Callback routine interruption	Value: `cancel`, `queue` Default: `queue`
`Callback`	Control action	Value: string
`CreateFcn`	Callback routine executed during object creation	Value: string
`DeleteFcn`	Callback routine executed during object deletion	Value: string
`Interruptible`	Callback routine interruption mode	Value: `on`, `off` Default: `on`
Controlling Access to Objects		
`HandleVisibility`	Whether handle is accessible from command line and GUIs	Value: `on`, `callback`, `off` Default: `on`

Example

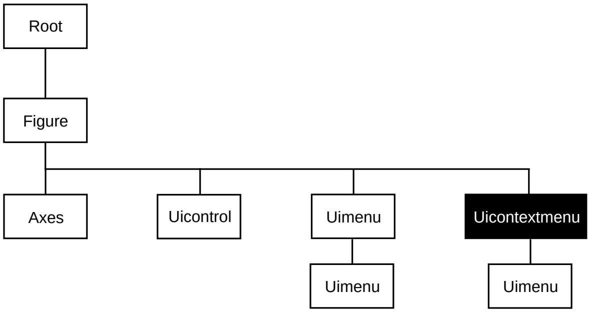
These statements define a context menu associated with a line. When the user extend-clicks anywhere on the line, the menu appears. Menu items enable the user to change the line style.

```
% Define the context menu
cmenu = uicontextmenu;
% Define the line and associate it with the context menu
hline = plot(1:10, 'UIContextMenu', cmenu);
% Define callbacks for context menu items
cb1 = ['set(hline, ''LineStyle'', ''--'')'];
cb2 = ['set(hline, ''LineStyle'', '':'')'];
cb3 = ['set(hline, ''LineStyle'', ''-'')'];
% Define the context menu items
item1 = uimenu(cmenu, 'Label', 'dashed', 'Callback', cb1);
item2 = uimenu(cmenu, 'Label', 'dotted', 'Callback', cb2);
item3 = uimenu(cmenu, 'Label', 'solid', 'Callback', cb3);
```

When the user extend-clicks on the line, the context menu appears, as shown in this figure:



Object Hierarchy



See Also

`uicontrol`, `uimenu`

Modifying Properties

You can set and query graphics object properties in two ways:

- The Property Editor is an interactive tool that enables you to see and change object property values.
- The `set` and `get` commands enable you to set and query the values of properties

To change the default value of properties see Settingcreating_plots Default Property Values.

Uicontextmenu Property Descriptions

`BusyAction` `cancel | {queue}`

Callback routine interruption. The `BusyAction` property enables you to control how MATLAB handles events that potentially interrupt executing callback routines. If a callback routine is executing, subsequently invoked callback routines always attempt to interrupt it. If the `Interruptible` property of the object whose callback is executing is set to `on` (the default), then interruption occurs at the next point where the event queue is processed. If the `Interruptible` property is `off`, the `BusyAction` property of the object whose callback is executing determines how MATLAB handles the event. The choices are:

- `cancel` – discard the event that attempted to execute a second callback routine.
- `queue` – queue the event that attempted to execute a second callback routine until the current callback finishes.

`ButtonDownFcn` string

This property has no effect on uicontextmenu objects.

`Callback` string

Control action. A routine that executes whenever you right-click on an object for which a context menu is defined. The routine executes immediately before the context menu is posted. Define this routine as a string that is a valid MATLAB expression or the name of an M-file. The expression executes in the MATLAB workspace.

`Children` matrix

The uimenus defined for the uicontextmenu.

Clipping {on} | off

This property has no effect on uicontextmenu objects.

CreateFcn string

Callback routine executed during object creation. This property defines a callback routine that executes when MATLAB creates a uicontextmenu object. You must define this property as a default value for uicontextmenus. For example, this statement:

```
set(0,'DefaultUicontextmenuCreateFcn',...
      'set(gcf,''IntegerHandle'',''off'')')
```

defines a default value on the root level that sets the figure `IntegerHandle` property to `off` whenever you create a uicontextmenu object. MATLAB executes this routine after setting all property values for the uicontextmenu. Setting this property on an existing uicontextmenu object has no effect.

The handle of the object whose `CreateFcn` is being executed is accessible only through the root `CallbackObject` property, which can be queried using `gcbo`.

DeleteFcn string

Delete uicontextmenu callback routine. A callback routine that executes when you delete the uicontextmenu object (e.g., when you issue a `delete` command or clear the figure containing the uicontextmenu). MATLAB executes the routine before destroying the object's properties so these values are available to the callback routine.

The handle of the object whose `DeleteFcn` is being executed is accessible only through the root `CallbackObject` property, which you can query using `gcbo`.

HandleVisibility {on} | callback | off

Control access to object's handle by command-line users and GUIs. This property determines when an object's handle is visible in its parent's list of children. `HandleVisibility` is useful for preventing command-line users from accidentally drawing into or deleting a figure that contains only user interface devices (such as a dialog box).

Handles are always visible when `HandleVisibility` is `on`.

Setting `HandleVisibility` to `callback` causes handles to be visible from within callback routines or functions invoked by callback routines, but not from within functions invoked from the command line. This provides a means to protect GUIs from command-line users, while allowing callback routines to have complete access to object handles.

Setting `HandleVisibility` to `off` makes handles invisible at all times. This may be necessary when a callback routine invokes a function that might potentially damage the GUI (such as evaluating a user-typed string), and so temporarily hides its own handles during the execution of that function.

When a handle is not visible in its parent's list of children, it cannot be returned by functions that obtain handles by searching the object hierarchy or querying handle properties. This includes `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close`.

When a handle's visibility is restricted using `callback` or `off`, the object's handle does not appear in its parent's `Children` property, figures do not appear in the root's `CurrentFigure` property, objects do not appear in the root's `CallbackObject` property or in the figure's `CurrentObject` property, and axes do not appear in their parent's `CurrentAxes` property.

You can set the root `ShowHiddenHandles` property to on to make all handles visible, regardless of their `HandleVisibility` settings (this does not affect the values of the `HandleVisibility` properties).

Handles that are hidden are still valid. If you know an object's handle, you can `set` and `get` its properties, and pass it to any function that operates on handles.

`HitTest` `{on} | off`

This property has no effect on uicontextmenu objects.

`Interruptible` `{on} | off`

Callback routine interruption mode. The `Interruptible` property controls whether a uicontextmenu callback routine can be interrupted by subsequently invoked callback routines. By default (on), execution of a callback routine can be interrupted.

Only callback routines defined for the `ButtonDownFcn` and `Callback` properties are affected by the `Interruptible` property. MATLAB checks for events that can interrupt a callback routine only when it encounters a `drawnow`, `figure`, `getframe`, `pause`, or `waitfor` command in the routine.

Parent handle

Uicontextmenu's parent. The handle of the uicontextmenu's parent object. The parent of a uicontextmenu object is the figure in which it appears. You can move a uicontextmenu object to another figure by setting this property to the handle of the new parent.

Position vector

Uicontextmenu's position. A two-element vector that defines the location of a context menu posted by setting the `Visible` property value to `on`. Specify `Position` as

```
[left bottom]
```

where vector elements represent the distance in pixels from the bottom left corner of the figure window to the top left corner of the context menu.

Selected `on | {off}`

This property has no effect on uicontextmenu objects.

SelectionHighlight `{on} | off`

This property has no effect on uicontextmenu objects.

Tag string

User-specified object label. The `Tag` property provides a means to identify graphics objects with a user-specified label. This is particularly useful when constructing interactive graphics programs that would otherwise need to define object handles as global variables or pass them as arguments between callback routines. You can define `Tag` as any string.

Type string

Class of graphics object. For uicontextmenu objects, `Type` is always the string `'uicontextmenu'`.

UIContextMenu handle

This property has no effect on uicontextmenus.

UserData matrix

User-specified data. Any data you want to associate with the uicontextmenu object. MATLAB does not use this data, but you can access it using `set` and `get`.

Visible on | {off}

Uicontextmenu visibility. The Visible property can be used in two ways:

- Its value indicates whether the context menu is currently posted. While the context menu is posted, the property value is on; when the context menu is not posted, its value is off.
- Its value can be set to on to force the posting of the context menu. Similarly, setting the value to off forces the context menu to be removed. When used in this way, the Position property determines the location of the posted context menu.

uicontrol

Purpose

Create user interface control object

Syntax

```
handle = uicontrol(parent)
handle = uicontrol(...,'PropertyName',PropertyValue,...)
```

Description

`uicontrol` creates uicontrol graphics objects (user interface controls). You implement graphical user interfaces using uicontrols. When selected, most uicontrol objects perform a predefined action. MATLAB supports numerous styles of uicontrols, each suited for a different purpose:

- Check boxes
- Editable text
- Frames
- List boxes
- Pop-up menus
- Push buttons
- Radio buttons
- Sliders
- Static text
- Toggle buttons

Check boxes generate an action when clicked on. These devices are useful when providing the user with a number of independent choices. To activate a check box, click the mouse button on the object. The state of the device is indicated on the display.

Editable text boxes are fields that enable users to enter or modify text values. Use editable text when you want text as input.

On Microsoft Windows systems, if an editable text box has focus, clicking on the menu bar does not cause the editable text callback routine to execute. However, it does cause execution on UNIX systems. Therefore, after clicking on the menu bar, the statement

```
get(edit_handle,'String')
```

does not return the current contents of the edit box on Microsoft Windows systems because MATLAB must execute the callback routine to update the

`String` property (even though the text string has changed on the screen). This behavior is consistent with the respective platform conventions.

Frames are boxes that visually enclose regions of a figure window. Frames can make a user interface easier to understand by visually grouping related controls. Frames have no callback routines associated with them. Only uicontrols can appear within frames.

Frames are opaque, not transparent, so the order you define uicontrols is important in determining whether uicontrols within a frame are covered by the frame or are visible. *Stacking order* determines the order objects are drawn: objects defined first are drawn first; objects defined later are drawn over existing objects. If you use a frame to enclose objects, you must define the frame before you define the objects.

List boxes display a list of items (defined using the `String` property) and enable users to select one or more items. The `Min` and `Max` properties control the selection mode. The `Value` property indicates selected entries and contains the indices into the list of strings; a vector value indicates multiple selections. MATLAB evaluates the list box's callback routine after any mouse button up event that changes the `Value` property. Therefore, you may need to add a "Done" button to delay action caused by multiple clicks on list items. List boxes differentiate between single and double clicks and set the figure `SelectionType` property to `normal` or `open` accordingly before evaluating the list box's `Callback` property.

Pop-up menus open to display a list of choices (defined using the `String` property) when pressed. When not open, a pop-up menu indicates the current choice. Pop-up menus are useful when you want to provide users with a number of mutually exclusive choices, but do not want to take up the amount of space that a series of radio buttons requires. You must specify a value for the `String` property.

Push buttons generate an action when pressed. To activate a push button, click the mouse button on the push button.

Radio buttons are similar to check boxes, but are intended to be mutually exclusive within a group of related radio buttons (i.e., only one is in a pressed state at any given time). To activate a radio button, click the mouse button on the object. The state of the device is indicated on the display. Note that your code can implement the mutually exclusive behavior of radio buttons.

Sliders accept numeric input within a specific range by enabling the user to move a sliding bar. Users move the bar by pressing the mouse button and dragging the pointer over the bar, or by clicking in the trough or on an arrow. The location of the bar indicates a numeric value, which is selected by releasing the mouse button. You can set the minimum, maximum, and current values of the slider.

Static text boxes display lines of text. Static text is typically used to label other controls, provide directions to the user, or indicate values associated with a slider. Users cannot change static text interactively and there is no way to invoke the callback routine associated with it.

Toggle buttons are controls that execute callbacks when clicked on and indicate their state, either on or off. Toggle buttons are useful for building toolbars.

Remarks

The `uicontrol` function accepts property name/property value pairs, structures, and cell arrays as input arguments and optionally returns the handle of the created object. You can also set and query property values after creating the object using the `set` and `get` functions.

Uicontrol objects are children of figures and therefore do not require an axes to exist when placed in a figure window.

Properties

This table lists all properties useful for `uicontrol` objects, grouping them by function. Each property name acts as a link to a description of the property.

Property Name	Property Description	Property Value
Controlling Style and Appearance		
`BackgroundColor`	Object background color	Value: `ColorSpec` Default: system dependent
`CData`	Truecolor image displayed on the control	Value: matrix
`ForegroundColor`	Color of text	Value: `ColorSpec` Default: `[0 0 0]`
`SelectionHighlight`	Object highlighted when selected	Value: `on`, `off` Default: `on`

Property Name	Property Description	Property Value
String	Uicontrol label, also list box and pop-up menu items	Value: string
Visible	Uicontrol visibility	Value: on, off Default: on
General Information About the Object		
Children	Uicontrol objects have no children	
Enable	Enable or disable the uicontrol	Value: on, inactive, off Default: on
Parent	Uicontrol object's parent	Value: scalar figure handle
Selected	Whether object is selected	Value: on, off Default: off
SliderStep	Slider step size	Value: two-element vector Default: [0.01 0.1]
Style	Type of uicontrol object	Value: pushbutton, togglebutton, radiobutton, checkbox, edit, text, slider, frame, listbox, popupmenu Default: pushbutton
Tag	User-specified object identifier	Value: string
TooltipString	Content of object's tooltip	Value: string
Type	Class of graphics object	Value: string (read-only) Default: uicontrol
UserData	User-specified data	Value: matrix
Controlling the Object Position		
Position	Size and location of uicontrol object	Value: position rectangle Default: [20 20 60 20]

Property Name	Property Description	Property Value
Units	Units to interpret position vector	Value: pixels, normalized, inches, centimeters, points, characters Default: pixels
Controlling Fonts and Labels		
FontAngle	Character slant	Value: normal, italic, oblique Default: normal
FontName	Font family	Value: string Default: system dependent
FontSize	Font size	Value: size in FontUnits Default: system dependent
FontUnits	Font size units	Value: points, normalized, inches, centimeters, pixels Default: points
FontWeight	Weight of text characters	Value: light, normal, demi, bold Default: normal
HorizontalAlignment	Alignment of label string	Value: left, center, right Default: depends on uicontrol object
String	Uicontrol object label, also list box and pop-up menu items	Value: string
Controlling Callback Routine Execution		
BusyAction	Callback routine interruption	Value: cancel, queue Default: queue
ButtonDownFcn	Button press callback routine	Value: string
Callback	Control action	Value: string

Property Name	Property Description	Property Value
CreateFcn	Callback routine executed during object creation	Value: string
DeleteFcn	Callback routine executed during object deletion	Value: string
Interruptible	Callback routine interruption mode	Value: on, off Default: on
UIContextMenu	Uicontextmenu object associated with the uicontrol	Value: handle
Information About the Current State		
ListboxTop	Index of top-most string displayed in list box	Value: scalar Default: [1]
Max	Maximum value (depends on uicontrol object)	Value: scalar Default: object dependent
Min	Minimum value (depends on uicontrol object)	Value: scalar Default: object dependent
Value	Current value of uicontrol object	Value: scalar or vector Default: object dependent
Controlling Access to Objects		
HandleVisibility	Whether handle is accessible from command line and GUIs	Value: on, callback, off Default: on
HitTest	Whether selectable by mouse click	Value: on, off Default: on

Examples

The following statement creates a push button that clears the current axes when pressed:

```
h = uicontrol('Style', 'pushbutton', 'String', 'Clear',...
    'Position', [20 150 100 70], 'Callback', 'cla');
```

You can create a uicontrol object that changes figure colormaps by specifying a pop-up menu and supplying an M-file name as the object's `Callback`:

```
hpop = uicontrol('Style', 'popup',...
       'String', 'hsv|hot|cool|gray',...
       'Position', [20 320 100 50],...
       'Callback', 'setmap');
```

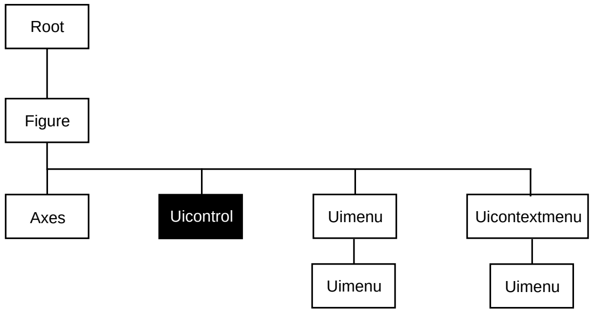
The above call to `uicontrol` defines four individual choices in the menu: `hsv`, `hot`, `cool`, and `gray`. You specify these choices with the `String` property, separating the choices with the "|" character.

The `Callback`, in this case `setmap`, is the name of an M-file that defines a more complicated set of instructions than a single MATLAB command. `setmap` contains these statements:

```
val = get(hpop,'Value');
if val == 1
    colormap(hsv)
elseif val == 2
    colormap(hot)
elseif val == 3
    colormap(cool)
elseif val == 4
    colormap(gray)
end
```

The `Value` property contains a number that indicates the selected choice. The choices are numbered sequentially from one to four. The `setmap` M-file can get and then test the contents of the `Value` property to determine what action to take.

Object Hierarchy



See Also

`textwrap`, `uimenu`

Uicontrol Properties

Modifying Properties

You can set and query graphics object properties in two ways:

- The Property Editor is an interactive tool that enables you to see and change object property values.
- The `set` and `get` commands enable you to set and query the values of properties

To change the default value of properties see Settingcreating_plots Default Property Values.

Uicontrol Property Descriptions

You can set default uicontrol properties on the root and figure levels:

```
set(0,'DefaultUicontrolProperty',PropertyValue...)
set(gcf,'DefaultUicontrolProperty',PropertyValue...)
```

where *`Property`* is the name of the uicontrol property whose default value you want to set and `PropertyValue` is the value you are specifying. Use `set` and `get` to access uicontrol properties.

Curly braces { } enclose the default value.

`BackgroundColor` `ColorSpec`

Object background color. The color used to fill the uicontrol rectangle. Specify a color using a three-element RGB vector or one of MATLAB's predefined names. The default color is determined by system settings. See `ColorSpec` for more information on specifying color.

`BusyAction` `cancel | {queue}`

Callback routine interruption. If a callback is executing and the user triggers an event (such as a mouse click) on an object for which a callback is defined, that callback attempts to interrupt the first callback. The first callback can be interrupted only at a `drawnow`, `figure`, `getframe`, `pause`, or `waitfor` command; if the callback does not contain any of these commands, it cannot be interrupted.

If the `Interruptible` property of the object whose callback is executing is `off` (the default value is on), the callback cannot be interrupted (except by certain callbacks; see the note below). The `BusyAction` property of the object whose callback is waiting to execute determines what happens to the callback:

- If the value is `queue`, the callback is added to the event queue and executes after the first callback finishes execution.
- If the value is `cancel`, the event is discarded and the callback is not executed.

Note If the interrupting callback is a `DeleteFcn` or `CreateFcn` callback or a figure's `CloseRequest` or `ResizeFcn` callback, it interrupts an executing callback regardless of the value of that object's `Interruptible` property. The interrupting callback starts execution at the next `drawnow`, `figure`, `getframe`, `pause`, or `waitfor` statement.

`ButtonDownFcn` string

Button press callback routine. A callback routine that executes whenever you press a mouse button while the pointer is in a five-pixel wide border around the uicontrol. When the uicontrol's `Enable` property is set to `inactive` or `off`, the `ButtonDownFcn` executes when you click the mouse in the five-pixel border or on the control itself. This is useful for implementing actions to interactively modify control object properties, such as size and position, when they are clicked on (using `selectmoveresize`, for example).

Define this routine as a string that is a valid MATLAB expression or the name of an M-file. The expression executes in the MATLAB workspace.

The `Callback` property defines the callback routine that executes when you activate the enabled uicontrol (e.g., click on a push button).

`Callback` string (GUIDE sets this property)

Control action. A routine that executes whenever you activate the uicontrol object (e.g., when you click on a push button or move a slider). Define this routine as a string that is a valid MATLAB expression or the name of an M-file. The expression executes in the MATLAB workspace.

To execute the callback routine for an editable text control, type in the desired text, then either:

- Move the focus off the object (click the mouse someplace else in the GUI),
- For a single line editable text box, press **Return**, or
- For a multiline editable text box, press **Ctl-Return**.

Callback routines defined for frames and static text do not execute because no action is associated with these objects.

CData matrix

Truecolor image displayed on control. A three-dimensional matrix of RGB values that defines a truecolor image displayed on either a push button or toggle button. Each value must be between 0.0 and 1.0.

Children matrix

The empty matrix; uicontrol objects have no children.

Clipping `{on} | off`

This property has no effect on uicontrols.

CreateFcn string

Callback routine executed during object creation. This property defines a callback routine that executes when MATLAB creates a uicontrol object. You must define this property as a default value for uicontrols. For example, this statement:

```
set(0,'DefaultUicontrolCreateFcn',...
      'set(gcf,''IntegerHandle'',''off'')')
```

defines a default value on the root level that sets the figure `IntegerHandle` property to `off` whenever you create a uicontrol object. MATLAB executes this routine after setting all property values for the uicontrol. Setting this property on an existing uicontrol object has no effect.

The handle of the object whose `CreateFcn` is being executed is accessible only through the root `CallbackObject` property, which can be queried using `gcbo`.

DeleteFcn string

Delete uicontrol callback routine. A callback routine that executes when you delete the uicontrol object (e.g., when you issue a `delete` command or clear the figure containing the uicontrol). MATLAB executes the routine before destroying the object's properties so these values are available to the callback routine.

The handle of the object whose `DeleteFcn` is being executed is accessible only through the root `CallbackObject` property, which you can query using `gcbo`.

Enable {on} | inactive | off

Enable or disable the uicontrol. This property controls how uicontrols respond to mouse button clicks, including which callback routines execute.

- on – The uicontrol is operational (the default).
- inactive – The uicontrol is not operational, but looks the same as when Enable is on.
- off – The uicontrol is not operational and its label (set by the string property) is grayed out.

When you left-click on a uicontrol whose Enable property is on, MATLAB performs these actions in this order:

1. Sets the figure's SelectionType property.
2. Executes the control's Callback routine.
3. Does not set the figure's CurrentPoint property and does not execute either the control's ButtonDownFcn or the figure's WindowButtonDownFcn callback.

When you left-click on a uicontrol whose Enable property is inactive or off, or when you right-click on a uicontrol whose Enable property has any value, MATLAB performs these actions in this order:

1. Sets the figure's SelectionType property.
2. Sets the figure's CurrentPoint property.
3. Executes the figure's WindowButtonDownFcn callback.
4. On a right-click, if the uicontrol is associated with a context menu, posts the context menu.
5. Executes the control's ButtonDownFcn callback.
6. Executes the selected context menu item's Callback routine.
7. Does not execute the control's Callback routine.

Setting this property to inactive or off enables you to implement object dragging or resizing using the ButtonDownFcn callback routine.

Extent position rectangle (read only)

Size of uicontrol character string. A four-element vector that defines the size and position of the character string used to label the uicontrol. It has the form:

```
[0,0,width,height]
```

The first two elements are always zero. `width` and `height` are the dimensions of the rectangle. All measurements are in units specified by the `Units` property.

Since the `Extent` property is defined in the same units as the uicontrol itself, you can use this property to determine proper sizing for the uicontrol with regard to its label. Do this by

- Defining the `String` property and selecting the font using the relevant properties.
- Getting the value of the `Extent` property.
- Defining the `width` and `height` of the `Position` property to be somewhat larger than the `width` and `height` of the `Extent`.

For multiline strings, the `Extent` rectangle encompasses all the lines of text. For single line strings, the Extent is returned as a single line, even if the string wraps when displayed on the control.

FontAngle `{normal} | italic | oblique`

Character slant. MATLAB uses this property to select a font from those available on your particular system. Setting this property to `italic` or `oblique` selects a slanted version of the font, when it is available on your system.

FontName string

Font family. The name of the font in which to display the `String`. To display and print properly, this must be a font that your system supports. The default font is system dependent.

To use a fixed-width font that looks good in any locale (and displays properly in Japan, where multibyte character sets are used), set `FontName` to the string `FixedWidth` (this string value is case sensitive):

```
set(uicontrol_handle, 'FontName', 'FixedWidth')
```

This parameter value eliminates the need to hard code the name of a fixed-width font, which may not display text properly on systems that do not use ASCII character encoding (such as in Japan). A properly written MATLAB application that needs to use a fixed-width font should set `FontName` to `FixedWidth` and rely on the root `FixedWidthFontName` property to be set correctly in the end user's environment.

End users can adapt a MATLAB application to different locales or personal environments by setting the root `FixedWidthFontName` property to the appropriate value for that locale from `startup.m`. Setting the root `FixedWidthFontName` property causes an immediate update of the display to use the new font.

`FontSize` size in `FontUnits`

Font size. A number specifying the size of the font in which to display the `String`, in units determined by the `FontUnits` property. The default point size is system dependent.

`FontUnits` `{points} | normalized | inches | centimeters | pixels`

Font size units. This property determines the units used by the `FontSize` property. `Normalized` units interpret `FontSize` as a fraction of the height of the uicontrol. When you resize the uicontrol, MATLAB modifies the screen `FontSize` accordingly. `pixels`, `inches`, `centimeters`, and `points` are absolute units (1 point = $^1/_{72}$ inch).

`FontWeight` `light | {normal} | demi | bold`

Weight of text characters. MATLAB uses this property to select a font from those available on your particular system. Setting this property to `bold` causes MATLAB to use a bold version of the font, when it is available on your system.

`ForegroundColor` `ColorSpec`

Color of text. This property determines the color of the text defined for the `String` property (the uicontrol label). Specify a color using a three-element RGB vector or one of MATLAB 's predefined names. The default text color is black. See `ColorSpec` for more information on specifying color.

`HandleVisibility` `{on} | callback | off`

Control access to object's handle by command-line users and GUIs. This property determines when an object's handle is visible in its parent's list of children. `HandleVisibility` is useful for preventing command-line users from accidentally drawing into or deleting a figure that contains only user interface devices (such as a dialog box).

Handles are always visible when `HandleVisibility` is `on`.

Setting `HandleVisibility` to `callback` causes handles to be visible from within callback routines or functions invoked by callback routines, but not from within functions invoked from the command line. This provides a means to protect GUIs from command-line users, while allowing callback routines to have complete access to object handles.

Setting `HandleVisibility` to `off` makes handles invisible at all times. This may be necessary when a callback routine invokes a function that might potentially damage the GUI (such as evaluating a user-typed string), and so temporarily hides its own handles during the execution of that function.

When a handle is not visible in its parent's list of children, it cannot be returned by functions that obtain handles by searching the object hierarchy or querying handle properties. This includes `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close`.

When a handle's visibility is restricted using `callback` or `off`, the object's handle does not appear in its parent's `Children` property, figures do not appear in the root's `CurrentFigure` property, objects do not appear in the root's `CallbackObject` property or in the figure's `CurrentObject` property, and axes do not appear in their parent's `CurrentAxes` property.

You can set the root `ShowHiddenHandles` property to on to make all handles visible, regardless of their `HandleVisibility` settings (this does not affect the values of the `HandleVisibility` properties).

Handles that are hidden are still valid. If you know an object's handle, you can `set` and `get` its properties, and pass it to any function that operates on handles.

`HitTest` `{on} | off`

Selectable by mouse click. This property has no effect on uicontrol objects.

`HorizontalAlignment` `left | {center} | right`

Horizontal alignment of label string. This property determines the justification of the text defined for the `String` property (the uicontrol label):

- `left` — Text is left justified with respect to the uicontrol.
- `center` — Text is centered with respect to the uicontrol.
- `right` — Text is right justified with respect to the uicontrol.

On Microsoft Windows systems, this property affects only `edit` and `text` uicontrols.

Interruptible {on} | off

Callback routine interruption mode. If a callback is executing and the user triggers an event (such as a mouse click) on an object for which a callback is defined, that callback attempts to interrupt the first callback. MATLAB processes the callbacks according to these factors:

- The Interruptible property of the object whose callback is executing
- Whether the executing callback contains drawnow, figure, getframe, pause, or waitfor statements
- The BusyAction property of the object whose callback is waiting to execute

If the Interruptible property of the object whose callback is executing is on (the default), the callback can be interrupted. The callback interrupts execution at the next drawnow, figure, getframe, pause, or waitfor statement, and processes the events in the event queue, which includes the waiting callback.

If the Interruptible property of the object whose callback is executing is off, the callback cannot be interrupted (except by certain callbacks; see the note below). The BusyAction property of the object whose callback is waiting to execute determines what happens to the callback.

Note If the interrupting callback is a DeleteFcn or CreateFcn callback or a figure's CloseRequest or ResizeFcn callback, it interrupts an executing callback regardless of the value of that object's Interruptible property. The interrupting callback starts execution at the next drawnow, figure, getframe, pause, or waitfor statement. A figure's WindowButtonDownFcn callback routine, or an object's ButtonDownFcn or Callback routine are processed according to the rules described above.

ListboxTop scalar

Index of top-most string displayed in list box. This property applies only to the listbox style of uicontrol. It specifies which string appears in the top-most position in a list box that is not large enough to display all list entries. ListboxTop is an index into the array of strings defined by the String property and must have a value between 1 and the number of strings. Noninteger values are fixed to the next lowest integer.

Max scalar

Maximum value. This property specifies the largest value allowed for the Value property. Different styles of uicontrols interpret Max differently:

- Check boxes – Max is the setting of the Value property while the check box is selected.
- Editable text – If Max – Min > 1, then editable text boxes accept multiline input. If Max – Min <= 1, then editable text boxes accept only single line input.
- List boxes – If Max – Min > 1, then list boxes allow multiple item selection. If Max – Min <= 1, then list boxes do not allow multiple item selection.
- Radio buttons – Max is the setting of the Value property when the radio button is selected.
- Sliders – Max is the maximum slider value and must be greater than the Min property. The default is 1.
- Toggle buttons – Max is the value of the Value property when the toggle button is selected. The default is 1.
- Frames, pop-up menus, push buttons, and static text do not use the Max property.

Min scalar

Minimum value. This property specifies the smallest value allowed for the Value property. Different styles of uicontrols interpret Min differently:

- Check boxes – Min is the setting of the Value property while the check box is not selected.
- Editable text – If Max – Min > 1, then editable text boxes accept multiline input. If Max – Min <= 1, then editable text boxes accept only single line input.
- List boxes – If Max – Min > 1, then list boxes allow multiple item selection. If Max – Min <= 1, then list boxes allow only single item selection.
- Radio buttons – Min is the setting of the Value property when the radio button is not selected.
- Sliders – Min is the minimum slider value and must be less than Max. The default is 0.
- Toggle buttons – Min is the value of the Value property when the toggle button is not selected. The default is 0.

- Frames, pop-up menus, push buttons, and static text do not use the `Min` property.

`Parent` handle

Uicontrol's parent. The handle of the uicontrol's parent object. The parent of a uicontrol object is the figure in which it appears. You can move a uicontrol object to another figure by setting this property to the handle of the new parent.

`Position` position rectangle

Size and location of uicontrol. The rectangle defined by this property specifies the size and location of the control within the figure window. Specify `Position` as

```
[left bottom width height]
```

`left` and `bottom` are the distance from the lower-left corner of the figure window to the lower-left corner of the uicontrol object. `width` and `height` are the dimensions of the uicontrol rectangle. All measurements are in units specified by the `Units` property.

On Microsoft Windows systems, the height of pop-up menus is automatically determined by the size of the font. The value you specify for the `height` of the `Position` property has no effect.

The `width` and `height` values determine the orientation of sliders. If `width` is greater than `height`, then the slider is oriented horizontally, If `height` is greater than `width`, then the slider is oriented vertically.

`Selected` `on | {off}`

Is object selected. When this property is `on`, MATLAB displays selection handles if the `SelectionHighlight` property is also `on`. You can, for example, define the `ButtonDownFcn` to set this property, allowing users to select the object with the mouse.

`SelectionHighlight` `{on} | off`

Object highlight when selected. When the `Selected` property is on, MATLAB indicates the selected state by drawing four edge handles and four corner handles. When `SelectionHighlight` is `off`, MATLAB does not draw the handles.

SliderStep `[min_step max_step]`

Slider step size. This property controls the amount the slider `Value` changes when you click the mouse on the arrow button (`min_step`) or on the slider trough (`max_step`). Specify `SliderStep` as a two-element vector; each value must be in the range `[0, 1]`. The actual step size is a function of the specified `SliderStep` and the total slider range (`Max – Min`). The default, `[0.01 0.10]`, provides a 1 percent change for clicks on the arrow button and a 10 percent change for clicks in the trough.

For example, if you create the following slider,

```
uicontrol('Style','slider','Min',1,'Max',7,...
          'SliderStep',[0.1 0.6])
```

clicking on the arrow button moves the indicator by,

```
0.1*(7-1)
ans =
     0.6000
```

and clicking in the trough moves the indicator by,

```
0.6*(7-1)
ans =
     3.6000
```

Note that if the specified step size moves the slider to a value outside the range, the indicator moves only to the `Max` or `Min` value.

See also the `Max`, `Min`, and `Value` properties.

String string

Uicontrol label, list box items, pop-up menu choices. For check boxes, editable text, push buttons, radio buttons, static text, and toggle buttons, the text displayed on the object. For list boxes and pop-up menus, the set of entries or items displayed in the object.

For uicontrol objects that display only one line of text, if the string value is specified as a cell array of strings or padded string matrix, only the first string of a cell array or of a padded string matrix is displayed; the rest are ignored. Vertical slash (`'|'`) characters are not interpreted as line breaks and instead show up in the text displayed in the uicontrol.

For multiple line editable text or static text controls, line breaks occur between each row of the string matrix, each cell of a cell array of strings, and after any `\n` characters embedded in the string. Vertical slash (`'|'`) characters are not interpreted as line breaks, and instead show up in the text displayed in the uicontrol.

For multiple items on a list box or pop-up menu, you can specify items as a cell array of strings, a padded string matrix, or within a string vector separated by vertical slash (`'|'`) characters.

For editable text, this property value is set to the string entered by the user.

`Style` `{pushbutton} | togglebutton | radiobutton | checkbox | edit | text | slider | frame | listbox | popupmenu`

Style of uicontrol object to create. The `Style` property specifies the kind of uicontrol to create. See the Description section for information on each type.

`Tag` string (GUIDE sets this property)

User-specified object label. The `Tag` property provides a means to identify graphics objects with a user-specified label. This is particularly useful when constructing interactive graphics programs that would otherwise need to define object handles as global variables or pass them as arguments between callback routines. You can define `Tag` as any string.

`TooltipString` string

Content of tooltip for object. The `TooltipString` property specifies the text of the tooltip associated with the uicontrol. When the user moves the mouse pointer over the control and leaves it there, the tooltip is displayed.

`Type` string (read only)

Class of graphics object. For uicontrol objects, `Type` is always the string `'uicontrol'`.

`UIContextMenu` handle

Associate a context menu with uicontrol. Assign this property the handle of a Uicontextmenu object. MATLAB displays the context menu whenever you right-click over the uicontrol. Use the `uicontextmenu` function to create the context menu.

Units `{pixels} | normalized | inches | centimeters | points | characters` (Guide default normalized)

Units of measurement. The units MATLAB uses to interpret the `Extent` and `Position` properties. All units are measured from the lower-left corner of the figure window. `Normalized` units map the lower-left corner of the figure window to (0,0) and the upper-right corner to (1.0,1.0). `pixels`, `inches`, `centimeters`, and `points` are absolute units (1 point = 1/72 inch). `Character` units are characters using the default system font; the width of one character is the width of the letter x, the height of one character is the distance between the baselines of two lines of text.

If you change the value of `Units`, it is good practice to return it to its default value after completing your computation so as not to affect other functions that assume `Units` is set to the default value.

UserData matrix

User-specified data. Any data you want to associate with the uicontrol object. MATLAB does not use this data, but you can access it using `set` and `get`.

Value scalar or vector

Current value of uicontrol. The uicontrol style determines the possible values this property can have:

- Check boxes set `Value` to `Max` when they are on (when selected) and `Min` when off (not selected).
- List boxes set `Value` to a vector of indices corresponding to the selected list entries, where 1 corresponds to the first item in the list.
- Pop-up menus set `Value` to the index of the item selected, where 1 corresponds to the first item in the menu. The Examples section shows how to use the `Value` property to determine which item has been selected.
- Radio buttons set `Value` to `Max` when they are on (when selected) and `Min` when off (not selected).
- Sliders set `Value` to the number indicated by the slider bar.
- Toggle buttons set `Value` to `Max` when they are down (selected) and `Min` when up (not selected).
- Editable text, frames, push buttons, and static text do not set this property.

Set the `Value` property either interactively with the mouse or through a call to the `set` function. The display reflects changes made to `Value`.

`Visible` `{on} | off`

Uicontrol visibility. By default, all uicontrols are visible. When set to `off`, the uicontrol is not visible, but still exists and you can query and set its properties.

uigetfile

Purpose

Interactively retrieve a filename

Syntax

```
uigetfile
uigetfile('FilterSpec')
uigetfile('FilterSpec','DialogTitle')
uigetfile('FilterSpec','DialogTitle',x,y)
[fname,pname] = uigetfile(...)
```

Description

`uigetfile` displays a dialog box used to retrieve a file. The dialog box lists the files and directories in the current directory.

`uigetfile('FilterSpec')` displays a dialog box that lists files in the current directory. `FilterSpec` determines the initial display of files and can be a full filename or include the * wildcard. For example, `'*.m'` lists all the MATLAB M-files. If `FilterSpec` is a cell array, the first column is use as the list of extensions, and the second column is used as the list of descriptions.

`uigetfile('FilterSpec','DialogTitle')` displays a dialog box that has the title `DialogTitle`.

`uigetfile('FilterSpec','DialogTitle',x,y)` positions the dialog box at position [x,y], where `x` and `y` are the distance in pixel units from the left and top edges of the screen. Note that some platforms may not support dialog box placement.

`[fname,pname] = uigetfile(...)` returns the name and path of the file selected in the dialog box. After you press the **Done** button, `fname` contains the name of the file selected and `pname` contains the name of the path selected. If you press the **Cancel** button or if an error occurs, `fname` and `pname` are set to `0`.

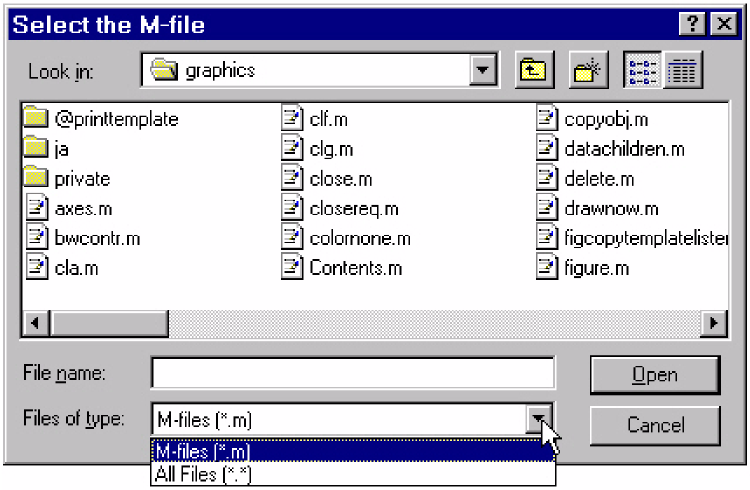
Remarks

If you select a file that does not exist, an error dialog appears. You can then enter another filename, or press the **Cancel** button.

Examples

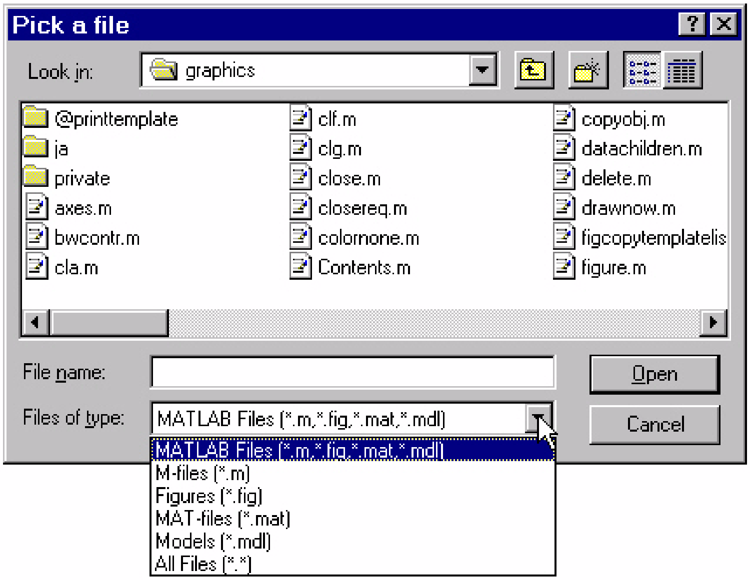
This statement displays a dialog box that enables you to retrieve a file. The statement lists all MATLAB M-files within a selected directory. The name and path of the selected file are returned in `fname` and `pname`. Note that `uigetfile` appends `All Files(*.*)` to the file types when `FilterSpec` is a string.

```
[fname,pname] = uigetfile('*.m','Select the M-file');
```



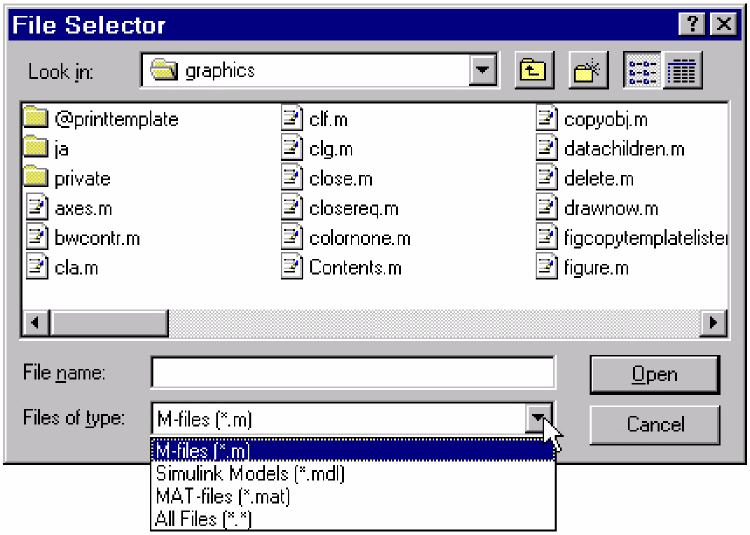
Use a cell array to specify a list of extensions and descriptions:

```
[filename, pathname] = uigetfile( ...
{'*.m;*.fig;*.mat;*.mdl','MATLAB Files (*.m,*.fig,*.mat,*.mdl)';
    '*.m',  'M-files (*.m)'; ...
    '*.fig','Figures (*.fig)'; ...
    '*.mat','MAT-files (*.mat)'; ...
    '*.mdl','Models (*.mdl)'; ...
    '*.*',  'All Files (*.*)'}, ...
    'Pick a file');
```



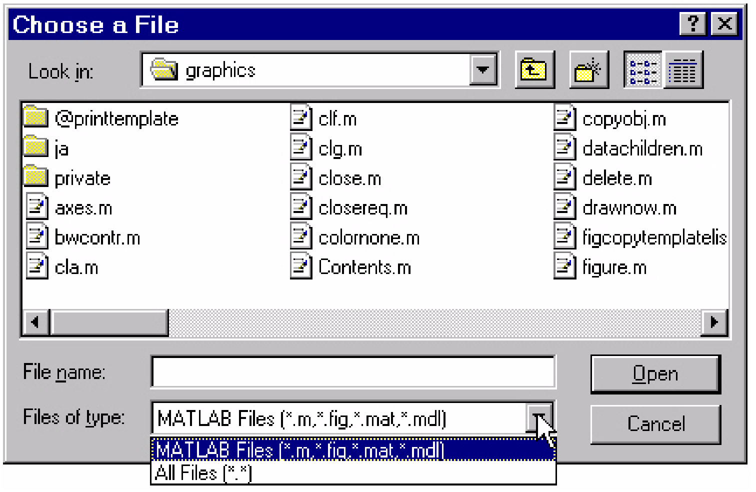
Separate multiple extensions with no descriptions with semi-colons.

```
[filename, pathname] = uigetfile(...
    {'*.m';'*.mdl';'*.mat';'*.*'},'File Selector');
```



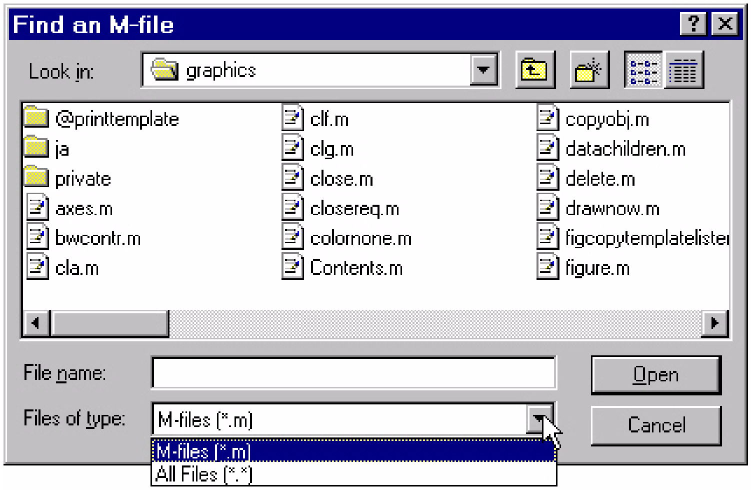
Associate multiple extensions with one description using the first column in the cell array for the file extensions and the second column as the description:

```
[filename, pathname] = uigetfile( ...
{'*.m;*.fig;*.mat;*.mdl','MATLAB Files (*.m,*.fig,*.mat,*.mdl)';
'*.*', 'All Files (*.*)'}, 'Choose a File');
```



This code checks for the existence of the file and returns a message about the success or failure of the open operation.

```
[filename, pathname] = uigetfile('*.m', 'Find an M-file');
if isequal(filename,0)|isequal(pathname,0)
    disp('File not found')
else
    disp(['File ', pathname, filename, ' found'])
end
```



The exact appearance of the dialog box depends on your windowing system.

See Also

`uiputfile`

uiimport

Purpose

Start the graphical user interface to import functions (Import Wizard)

Syntax

```
uiimport
uiimport(filename)
uiimport('-file')
uiimport('-pastespecial')
S = uiimport(...)
```

Description

`uiimport` starts the Import Wizard in the current directory, presenting options to load data from a file or the clipboard.

`uiimport(filename)` starts the Import Wizard, opening the file specified in `filename`. The Import Wizard displays a preview of the data in the file.

`uiimport('-file')` works as above but presents the file selection dialog first.

`uiimport('-pastespecial')` works as above but presents the clipboard contents first.

`S = uiimport(...)` works as above with resulting variables stored as fields in the struct `S`.

Note For ASCII data, you must verify that the Import Wizard correctly identified the column delimiter.

See Also

`load`, `clipboard`

Purpose Create menus on figure windows

Syntax

```
uimenu('PropertyName',PropertyValue,...)
uimenu(parent,'PropertyName',PropertyValue,...)
handle = uimenu('PropertyName',PropertyValue,...)
handle = uimenu(parent,'PropertyName',PropertyValue,...)
```

Description

`uimenu` creates a hierarchy of menus and submenus that are displayed in the figure window's menu bar. You can also use `uimenu` to create menu items for context menus.

`handle = uimenu('`*`PropertyName`*`',PropertyValue,...)` creates a menu in the current figure's menu bar using the values of the specified properties and assigns the menu handle to `handle`.

`handle = uimenu(parent,'`*`PropertyName`*`',PropertyValue,...)` creates a submenu of a parent menu or a menu item on a context menu specified by `parent` and assigns the menu handle to `handle`. If `parent` refers to a figure instead of another uimenu object or a Uicontextmenu, MATLAB creates a new menu on the referenced figure's menu bar.

Remarks

MATLAB adds the new menu to the existing menu bar. Each menu choice can itself be a menu that displays its submenu when selected.

`uimenu` accepts property name/property value pairs, as well as structures and cell arrays of properties as input arguments. The uimenu `Callback` property defines the action taken when you activate the menu item. `uimenu` optionally returns the handle to the created uimenu object.

Uimenus only appear in figures whose `WindowStyle` is `normal`. If a figure containing uimenu children is changed to `WindowStyle` `modal`, the uimenu children still exist and are contained in the `Children` list of the figure, but are not displayed until the `WindowStyle` is changed to `normal`.

The value of the figure `MenuBar` property affects the location of the uimenu on the figure menu bar. When `MenuBar` is `figure`, a set of built-in menus precedes the uimenus on the menu bar (MATLAB controls the built-in menus and their handles are not available to the user). When `MenuBar` is `none`, uimenus are the only items on the menu bar (that is, the built-in menus do not appear).

You can set and query property values after creating the menu using `set` and `get`.

Properties

This table lists all properties useful to `uimenu` objects, grouping them by function. Each property name acts as a link to a description of the property.

Property Name	Property Description	Property Value
Controlling Style and Appearance		
`Checked`	Menu check indicator	Value: `on`, `off` Default: `off`
`ForegroundColor`	Color of text	Value: `ColorSpec` Default: `[0 0 0]`
`Label`	Menu label	Value: string
`SelectionHighlight`	Object highlighted when selected	Value: `on`, `off` Default: `on`
`Separator`	Separator line mode	Value: `on`, `off` Default: `off`
`Visible`	Uimenu visibility	Value: `on`, `off` Default: `on`
General Information About the Object		
`Accelerator`	Keyboard equivalent	Value: character
`Children`	Handles of submenus	Value: vector of handles
`Enable`	Enable or disable the uimenu	Value: `on`, `off` Default: `on`
`Parent`	Uimenu object's parent	Value: handle
`Tag`	User-specified object identifier	Value: string
`Type`	Class of graphics object	Value: string (read-only) Default: `uimenu`

Property Name	Property Description	Property Value
`UserData`	User-specified data	Value: matrix
Controlling the Object Position		
`Position`	Relative uimenu position	Value: scalar Default: `[1]`
Controlling Callback Routine Execution		
`BusyAction`	Callback routine interruption	Value: `cancel`, `queue` Default: `queue`
`ButtonDownFcn`	Button press callback routine	Value: string
`Callback`	Control action	Value: string
`CreateFcn`	Callback routine executed during object creation	Value: string
`DeleteFcn`	Callback routine executed during object deletion	Value: string
`Interruptible`	Callback routine interruption mode	Value: `on`, `off` Default: `on`
Controlling Access to Objects		
`HandleVisibility`	Whether handle is accessible from command line and GUIs	Value: `on`, `callback`, `off` Default: `on`
`HitTest`	Whether selectable by mouse click	Value: `on`, `off` Default: `on`

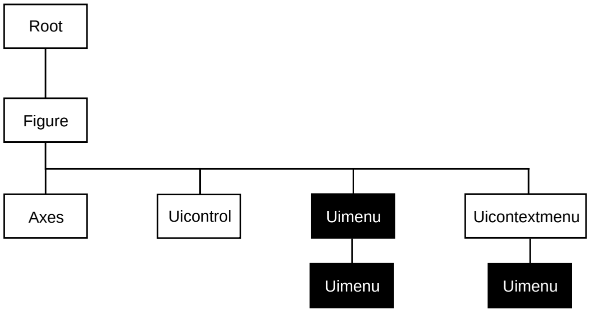
Examples

This example creates a menu labeled **Workspace** whose choices allow users to create a new figure window, save workspace variables, and exit out of MATLAB. In addition, it defines an accelerator key for the Quit option.

```
f = uimenu('Label','Workspace');
    uimenu(f,'Label','New Figure','Callback','figure');
    uimenu(f,'Label','Save','Callback','save');
```

```
uimenu(f,'Label','Quit','Callback','exit',...
       'Separator','on','Accelerator','Q');
```

Object Hierarchy



See Also

`uicontrol`, `uicontextmenu`, `gcbo`, `set`, `get`, `figure`

Modifying Properties

You can set and query graphics object properties in two ways:

- The Property Editor is an interactive tool that enables you to see and change object property values.
- The `set` and `get` commands enable you to set and query the values of properties

To change the default value of properties see Settingcreating_plots Default Property Values.

Uimenu Properties

This section lists property names along with the type of values each accepts. Curly braces { } enclose default values.

You can set default uimenu properties on the figure and root levels:

```
set(0,'DefaultUimenuPropertyName',PropertyValue...)
set(gcf,'DefaultUimenuPropertyName',PropertyValue...)
set(menu_handle,'DefaultUimenuProperty',PropertyValue...)
```

Where *`PropertyName`* is the name of the uimenu property and `PropertyValue` is the value you are specifying. Use `set` and `get` to access uimenu properties.

`Accelerator` character

Keyboard equivalent. A character specifying the keyboard equivalent for the menu item. This allows users to select a particular menu choice by pressing the specified character in conjunction with another key, instead of selecting the menu item with the mouse. The key sequence is platform specific:

- For Microsoft Windows systems, the sequence is **Ctrl-**`Accelerator`. These keys are reserved for default menu items: c, v, and x.
- For UNIX systems, the sequence is **Ctrl-**`Accelerator`. These keys are reserved for default menu items: o, p, s, and w.

You can define an accelerator only for menu items that do not have children menus. Accelerators work only for menu items that directly execute a callback routine, not items that bring up other menus.

Note that the menu item does not have to be displayed (e.g., a submenu) for the accelerator key to work. However, the window focus must be in the figure when the key sequence is entered.

BusyAction `cancel | {queue}`

Callback routine interruption. If a callback is executing and the user triggers an event (such as a mouse click) on an object for which a callback is defined, that callback attempts to interrupt the first callback. The first callback can be interrupted only at a `drawnow`, `figure`, `getframe`, `pause`, or `waitfor` command; if the callback does not contain any of these commands, it cannot be interrupted.

If the `Interruptible` property of the object whose callback is executing is `off` (the default value is `on`), the callback cannot be interrupted (except by certain callbacks; see the note below). The `BusyAction` property of the object whose callback is waiting to execute determines what happens to the callback:

- If the value is `queue`, the callback is added to the event queue and executes after the first callback finishes execution.
- If the value is `cancel`, the event is discarded and the callback is not executed.

Note If the interrupting callback is a `DeleteFcn` or `CreateFcn` callback or a figure's `CloseRequest` or `ResizeFcn` callback, it interrupts an executing callback regardless of the value of that object's `Interruptible` property. The interrupting callback starts execution at the next `drawnow`, `figure`, `getframe`, `pause`, or `waitfor` statement.

ButtonDownFcn string

The button down function has no effect on uimenu objects.

Callback string

Menu action. A callback routine that executes whenever you select the menu. Define this routine as a string that is a valid MATLAB expression or the name of an M-file. The expression executes in the MATLAB workspace.

A menu with children (submenus) executes its callback routine before displaying the submenus. A menu without children executes its callback routine when you *release* the mouse button (i.e., on the button up event).

Checked on | {off}

Menu check indicator. Setting this property to on places a check mark next to the corresponding menu item. Setting it to off removes the check mark. You can use this feature to create menus that indicate the state of a particular option. Note that there is no formal mechanism for indicating that an unchecked menu item will become checked when selected. Also, this property does not check top level menus or submenus, although you can change the value of the property for these menus.

Children vector of handles

Handles of submenus. A vector containing the handles of all children of the uimenu object. The children objects of uimenus are other uimenus, which function as submenus. You can use this property to re-order the menus.

Clipping {on} | off

Clipping has no effect on uimenu objects.

CreateFcn string

Callback routine executed during object creation. This property defines a callback routine that executes when MATLAB creates a uimenu object. You must define this property as a default value for uimenus. For example, the statement,

```
set(0,'DefaultUimenuCreateFcn','set(gcf,''IntegerHandle'',...
''off''’))
```

defines a default value on the root level that sets the figure IntegerHandle property to off whenever you create a uimenu object. Setting this property on an existing uimenu object has no effect. MATLAB executes this routine after setting all property values for the uimenu.

The handle of the object whose CreateFcn is being executed is accessible only through the root CallbackObject property, which can be queried using gcbo.

DeleteFcn string

Delete uimenu callback routine. A callback routine that executes when you delete the uimenu object (e.g., when you issue a delete command or cause the figure containing the uimenu to reset). MATLAB executes the routine before destroying the object’s properties so these values are available to the callback routine.

The handle of the object whose `DeleteFcn` is being executed is accessible only through the root `CallbackObject` property, which is more simply queried using `gcbo`.

`Enable` `{on} | off`

Enable or disable the uimenu. This property controls whether a menu item can be selected. When not enabled (set to `off`), the menu `Label` appears dimmed, indicating the user cannot select it.

`ForegroundColor` `ColorSpec` X-Windows only

Color of menu label string. This property determines color of the text defined for the `Label` property. Specify a color using a three-element RGB vector or one of MATLAB's predefined names. The default text color is black. See `ColorSpec` for more information on specifying color.

`HandleVisibility` `{on} | callback | off`

Control access to object's handle by command-line users and GUIs. This property determines when an object's handle is visible in its parent's list of children. `HandleVisibility` is useful for preventing command-line users from accidentally drawing into or deleting a figure that contains only user interface devices (such as a dialog box).

Handles are always visible when `HandleVisibility` is `on`.

Setting `HandleVisibility` to `callback` causes handles to be visible from within callback routines or functions invoked by callback routines, but not from within functions invoked from the command line. This provide a means to protect GUIs from command-line users, while allowing callback routines to have complete access to object handles.

Setting `HandleVisibility` to `off` makes handles invisible at all times. This may be necessary when a callback routine invokes a function that might potentially damage the GUI (such as evaluating a user-typed string), and so temporarily hides its own handles during the execution of that function.

When a handle is not visible in its parent's list of children, it cannot be returned by functions that obtain handles by searching the object hierarchy or querying handle properties. This includes `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close`.

When a handle's visibility is restricted using `callback` or `off`, the object's handle does not appear in its parent's `Children` property, figures do not appear in the root's `CurrentFigure` property, objects do not appear in the root's `CallbackObject` property or in the figure's `CurrentObject` property, and axes do not appear in their parent's `CurrentAxes` property.

You can set the root `ShowHiddenHandles` property to on to make all handles visible, regardless of their `HandleVisibility` settings (this does not affect the values of the `HandleVisibility` properties).

Handles that are hidden are still valid. If you know an object's handle, you can `set` and `get` its properties, and pass it to any function that operates on handles.

`HitTest` `{on} | off`

Selectable by mouse click. This property has no effect on uimenu objects.

`Interruptible` `{on} | off`

Callback routine interruption mode. If a callback is executing and the user triggers an event (such as a mouse click) on an object for which a callback is defined, that callback attempts to interrupt the first callback. MATLAB processes the callbacks according to these factors:

- The `Interruptible` property of the object whose callback is executing
- Whether the executing callback contains `drawnow`, `figure`, `getframe`, `pause`, or `waitfor` statements
- The `BusyAction` property of the object whose callback is waiting to execute

If the `Interruptible` property of the object whose callback is executing is `on` (the default), the callback can be interrupted. The callback interrupts execution at the next `drawnow`, `figure`, `getframe`, `pause`, or `waitfor` statement, and processes the events in the event queue, which includes the waiting callback.

If the `Interruptible` property of the object whose callback is executing is `off`, the callback cannot be interrupted (except by certain callbacks; see the note below). The `BusyAction` property of the object whose callback is waiting to execute determines what happens to the callback.

Note If the interrupting callback is a `DeleteFcn` or `CreateFcn` callback or a figure's `CloseRequest` or `ResizeFcn` callback, it interrupts an executing callback regardless of the value of that object's `Interruptible` property. The interrupting callback starts execution at the next `drawnow`, `figure`, `getframe`, `pause`, or `waitfor` statement. A figure's `WindowButtonDownFcn` callback routine, or an object's `ButtonDownFcn` or `Callback` routine are processed according to the rules described above.

`Label` string

Menu label. A string specifying the text label on the menu item. You can specify a mnemonic using the "&" character. Whatever character follows the "&" in the string appears underlined and selects the menu item when you type that character while the menu is visible. The "&" character is not displayed. To display the "&" character in a label, use two "&" characters in the string:

`'O&pen selection'` yields **Open selection**

`'Save && Go'` yields **Save & Go**

`Parent` handle

Uimenu's parent. The handle of the uimenu's parent object. The parent of a uimenu object is the figure on whose menu bar it displays, or the uimenu of which it is a submenu. You can move a uimenu object to another figure by setting this property to the handle of the new parent.

`Position` scalar

Relative menu position. The value of `Position` indicates placement on the menu bar or within a menu. Top-level menus are placed from left to right on the menu bar according to the value of their `Position` property, with 1 representing the left-most position. The individual items within a given menu are placed from top to bottom according to the value of their `Position` property, with 1 representing the top-most position.

`Selected` `on | {off}`

This property is not used for uimenu objects.

SelectionHighlight `on | off`

This property is not used for uimenu objects.

Separator `on | {off}`

Separator line mode. Setting this property to `on` draws a dividing line above the menu item.

Tag string

User-specified object label. The `Tag` property provides a means to identify graphics objects with a user-specified label. This is particularly useful when constructing interactive graphics programs that would otherwise need to define object handles as global variables or pass them as arguments between callback routines. You can define `Tag` as any string.

Type string (read only)

Class of graphics object. For uimenu objects, `Type` is always the string `'uimenu'`.

UserData matrix

User-specified data. Any matrix you want to associate with the uimenu object. MATLAB does not use this data, but you can access it using the `set` and `get` commands.

Visible `{on} | off`

Uimenu visibility. By default, all uimenus are visible. When set to `off`, the uimenu is not visible, but still exists and you can query and set its properties.

uint8, uint16, uint32

Purpose

Convert to unsigned integer

Syntax

```
i = uint8(x)
i = uint16(x)
i = uint32(x)
```

Description

`i = uint*(x)` converts the vector `x` into an unsigned integer. `x` can be any numeric object (such as a `double`). The results of a `uint*` operation are shown in the next table.

Operation	Output Range	Output Type	Bytes per Element	Output Class
`uint8`	0 to 255	Unsigned 8-bit integer	1	`uint8`
`uint16`	0 to 65535	Unsigned 16-bit integer	2	`uint16`
`uint32`	0 to 4294967295	Unsigned 32-bit integer	4	`uint32`

A value of `x` above or below the range for a class is mapped to one of the endpoints of the range. If `x` is already an unsigned integer of the same class, `uint*` has no effect.

The `uint*` class is primarily meant to store integer values. Most operations that manipulate arrays without changing their elements are defined (examples are `reshape`, `size`, the logical and relational operators, subscripted assignment, and subscripted reference). No math operations except for `sum` are defined for `uint*` since such operations are ambiguous on the boundary of the set (for example they could wrap or truncate there). You can define your own methods for `uint*` (as you can for any object) by placing the appropriately named method in an `@uint*` directory within a directory on your path.

Type `help datatypes` for the names of the methods you can overload.

See Also

`double`, `int8`, `int16`, `int32`, `single`

Purpose

Interactively select a file for writing

Syntax

```
uiputfile
uiputfile('InitFile')
uiputfile('InitFile','DialogTitle')
uiputfile('InitFile','DialogTitle',x,y)
[fname,pname] = uiputfile(...)
```

Description

`uiputfile` displays a dialog box used to select a file for writing. The dialog box lists the files and directories in the current directory.

`uiputfile('InitFile')` displays a dialog box that contains a list of files in the current directory determined by `InitFile`. `InitFile` is a full filename or includes the * wildcard. For example, specifying `'*.m'` (the default) causes the dialog box list to show only MATLAB M-files.

`uiputfile('InitFile','DialogTitle')` displays a dialog box that has the title `DialogTitle`.

`uiputfile('InitFile','DialogTitle',x,y)` positions the dialog box at screen position [x,y], where `x` and `y` are the distance in pixel units from the left and top edges of the screen. Note that positioning may not work on all platforms.

`[fname,pname] = uiputfile(...)` returns the name and path of the file selected in the dialog box. If you press the **Cancel** button or an error occurs, `fname` and `pname` are set to `0`.

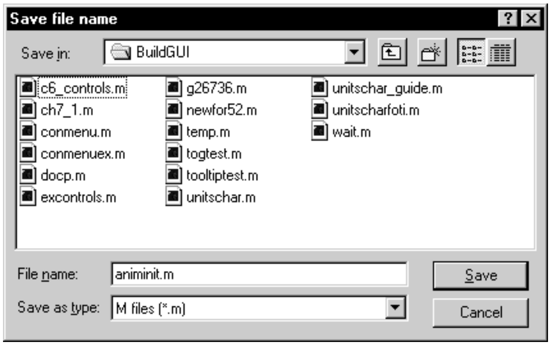
Remarks

If you select a file that already exists, a prompt asks whether you want to overwrite the file. If you choose to, the function successfully returns but does not delete the existing file (which is the responsibility of the calling routines). If you select **Cancel** in response to the prompt, the function returns control back to the dialog box so you can enter another filename.

Examples

This statement displays a dialog box titled `'Save file name'` (the exact appearance of the dialog box depends on your windowing system) with the filename `animinit.m`.

```
[newfile,newpath] = uiputfile('animinit.m','Save file name');
```



Microsoft
Windows

See Also

`uigetfile`

Purpose

Control program execution

Syntax

```
uiwait(h)
uiwait
uiresume(h)
```

Description

The `uiwait` and `uiresume` functions block and resume MATLAB program execution.

`uiwait` blocks execution until `uiresume` is called or the current figure is deleted. This syntax is the same as `uiwait(gcf)`.

`uiwait(h)` blocks execution until `uiresume` is called or the figure `h` is deleted.

`uiresume(h)` resumes the M-file execution that `uiwait` suspended.

Remarks

When creating a dialog, you should have a uicontrol with a callback that calls `uiresume` or a callback that destroys the dialog box. These are the only methods that resume program execution after the `uiwait` function blocks execution.

`uiwait` is a convenient way to use the `waitfor` command. You typically use it in conjunction with a dialog box. It provides a way to block the execution of the M-file that created the dialog, until the user responds to the dialog box. When used in conjunction with a modal dialog, `uiwait/uiresume` can block the execution of the M-file *and* restrict user interaction to the dialog only.

See Also

`uicontrol`, `uimenu`, `waitfor`, `figure`, `dialog`

uisetcolor

Purpose Set an object's `ColorSpec` from a dialog box interactively

Syntax

```
c = uisetcolor(h_or_c, 'DialogTitle');
```

Description `uisetcolor` displays a dialog box for the user to fill in, then applies the selected color to the appropriate property of the graphics object identified by the first argument.

`h_or_c` can be either a handle to a graphics object or an RGB triple. If you specify a handle, it must specify a graphics object that have a `Color` property. If you specify a color, it must be a valid RGB triple (e.g., [1 0 0] for red). The color specified is used to initialize the dialog box. If no initial RGB is specified, the dialog box initializes the color to black.

`DialogTitle` is a string that is used as the title of the dialog box.

`c` is the RGB value selected by the user. If the user presses **Cancel** from the dialog box, or if any error occurs, `c` is set to the input RGB triple, if provided; otherwise, it is set to 0.

See Also `ColorSpec`

Purpose Modify font characteristics for objects interactively

Syntax

```
uisetfont
uisetfont(h)
uisetfont(S)
uisetfont(h,'DialogTitle')
uisetfont(S,'DialogTitle')
S = uisetfont(...)
```

Description `uisetfont` enables you to change font properties (`FontName`, `FontUnits`, `FontSize`, `FontWeight`, and `FontAngle`) for a text, axes, or uicontrol object. The function returns a structure consisting of font properties and values. You can specify an alternate title for the dialog box.

`uisetfont` displays the dialog box and returns the selected font properties.

`uisetfont(h)` displays a dialog box, initializing the font property values with the values of those properties for the object whose handle is `h`. Selected font property values are applied to the current object. If a second argument is supplied, it specifies a name for the dialog box.

`uisetfont(S)` displays a dialog box, initializing the font property values with the values defined for the specified structure (`S`). `S` must define legal values for one or more of these properties: `FontName`, `FontUnits`, `FontSize`, `FontWeight`, and `FontAngle` and the field names must match the property names exactly. If other properties are defined, they are ignored. If a second argument is supplied, it specifies a name for the dialog box.

`uisetfont('DialogTitle')` displays a dialog box with the title `DialogTitle` and returns the values of the font properties selected in the dialog box.

If a left-hand argument is specified, the properties `FontName`, `FontUnits`, `FontSize`, `FontWeight`, and `FontAngle` are returned as fields in a structure. If the user presses **Cancel** from the dialog box or if an error occurs, the output value is set to 0.

Example These statements create a text object, then display a dialog box (labeled Update Font) that enables you to change the font characteristics:

```
h = text(.5,.5,'Figure Annotation');
uisetfont(h,'Update Font')
```

These statements create two push buttons, then set the font properties of one based on the values set for the other:

```
% Create push button with string ABC
c1 = uicontrol('Style', 'pushbutton', ...
     'Position', [10 10 100 20], 'String', 'ABC');
% Create push button with string XYZ
c2 = uicontrol('Style', 'pushbutton', ...
     'Position', [10 50 100 20], 'String', 'XYZ');
% Display set font dialog box for c1, make selections, save to d
d = uisetfont(c1)
% Apply those settings to c2
set(c2, d)
```

See Also

`axes`, `text`, `uicontrol`

Purpose Undo previous checkout from source control system

Graphical Interface As an alternative to the undocheckout function, use **Source Control Undo Checkout** in the Editor, Simulink, or Stateflow **File** menu.

Syntax

```
undocheckout('filename')
undocheckout({'filename1','filename2','filename3', ...})
```

Description `undocheckout('filename')` makes the file `filename` available for checkout, where `filename` does not reflect any of the changes you made after you last checked it out. `filename` must be the full pathname for the file.

`undocheckout({'filename1','filename2','filename3', ...})` makes the `filename1` through `filenamen` available for checkout, where the files do not reflect any of the changes you made after you last checked them out. Use the full pathnames for the files.

Examples Typing

```
undocheckout({'/matlab/mymfiles/clock.m', ...
'/matlab/mymfiles/calendar.m'})
```

undoes the checkouts of `/matlab/mymfiles/clock.m` and `/matlab/mymfiles/calendar.m` from the source control system.

See Also `checkin`, `checkout`

union

Purpose

Set union of two vectors

Syntax

```
c = union(A,B)
c = union(A,B,'rows')
[c,ia,ib] = union(...)
```

Description

`c = union(A,B)` returns the combined values from `A` and B but with no repetitions. The resulting vector is sorted in ascending order. In set theoretic terms, `c = A` $\cup$ B. A and B can be cell arrays of strings.

`c = union(A,B,'rows')` when `A` and `B` are matrices with the same number of columns returns the combined rows from `A` and `B` with no repetitions.

`[c,ia,ib] = union(...)` also returns index vectors `ia` and `ib` such that `c = a(ia)` $\cup$ `b(ib)`, or for row combinations, `c = a(ia,:)` $\cup$ `b(ib,:)`. If a value appears in both `a` and `b`, `union` indexes its occurrence in `b`. If a value appears more than once in `b` or in `a` (but not in `b`), `union` indexes the last occurrence of the value.

Examples

```
a = [-1 0 2 4 6];
b = [-1 0 1 3];
[c,ia,ib] = union(a,b);
c =

    -1     0     1     2     3     4     6

ia =

     3     4     5

ib =

     1     2     3     4
```

See Also

`intersect`, `setdiff`, `setxor`, `unique`

Purpose Unique elements of a vector

Syntax

```
b = unique(A)
b = unique(A,'rows')
[b,m,n] = unique(...)
```

Description `b = unique(A)` returns the same values as in `A` but with no repetitions. The resulting vector is sorted in ascending order. `A` can be a cell array of strings.

`b = unique(A,'rows')` returns the unique rows of `A`.

`[b,m,n] = unique(...)` also returns index vectors `m` and `n` such that `b = a(m)` and `a = b(n)`. Each element of `m` is the greatest subscript such that `b = a(m)`. For row combinations, `b = a(m,:)` and `a = b(n,:)`.

Examples

```
a = [1 1 5 6 2 3 3 9 8 6 2 4]
a =
1     1     5     6     2     3     3     9     8     6     2     4

[b,m,n] = unique(a)
b =
     1     2     3     4     5     6     8     9
m =
     2    11     7    12     3    10     9     8
n =
1     1     5     6     2     3     3     8     7     6     2     4

a(m)
ans =
     1     2     3     4     5     6     8     9

b(n)
ans =
1     1     5     6     2     3     3     9     8     6     2     4
```

Because `NaN`s are not equal to each other, `unique` treats them as unique elements.

```
unique([1 1 NaN NaN])
ans =
     1 NaN NaN
```

See Also `intersect`, `ismember`, `setdiff`, `setxor`, `union`

Purpose

Execute a UNIX command and return result

Syntax

```
unix command
status = unix('command')
[status,result] = unix('command')
[status,result] = unix('command','-echo')
```

Description

`unix command` calls upon the UNIX operating system to execute the given command.

`status = unix('command')` returns completion status to the `status` variable.

`[status, result] = unix('command')` returns the standard output to the `result` variable, in addition to completion status.

`[status,result] = unix('command','-echo')` forces the output to the Command Window, even though it is also being assigned into a variable.

Examples

The following example lists all users that are currently logged in. It returns a zero (success) in `s` and a string containing the list of users in `w`.

```
[s,w] = unix('who');
```

The next example returns a nonzero value in `s` to indicate failure and returns an error message in `w` because `why` is not a UNIX command.

```
[s,w] = unix('why')
s =
    1
w =
why: Command not found.
```

When including the `-echo` flag, MATLAB displays the results of the command in the Command Window as it executes as well as assigning the results to the return variable, `w`.

```
[s,w] = unix('who','-echo');
```

See Also

`Special Characters`

unmkpp

Purpose Piecewise polynomial details

Syntax

```
[breaks,coefs,l,k,d] = unmkpp(pp)
```

Description `[breaks,coefs,l,k,d] = unmkpp(pp)` extracts, from the piecewise polynomial `pp`, its breaks `breaks`, coefficients `coefs`, number of pieces `l`, order `k`, and dimension `d` of its target. Create `pp` using `spline` or the spline utility `mkpp`.

Examples This example creates a description of the quadratic polynomial

$$\frac{-x^2}{4} + x$$

as a piecewise polynomial `pp`, then extracts the details of that description.

```
pp = mkpp([-8 -4],[-1/4 1 0]);
[breaks,coefs,l,k,d] = unmkpp(pp)

breaks =
    -8    -4

coefs =
   -0.2500    1.0000         0

l =
     1

k =
     3

d =
     1
```

See Also `mkpp`, `ppval`, `spline`

Purpose

Correct phase angles

Syntax

```
Q = unwrap(P)
Q = unwrap(P,tol)
Q = unwrap(P,[],dim)
Q = unwrap(P,tol,dim)
```

Description

`Q = unwrap(P)` corrects the radian phase angles in array `P` by adding multiples of $\pm 2\pi$ when absolute jumps between consecutive array elements are greater than π radians. If `P` is a matrix, `unwrap` operates columnwise. If `P` is a multidimensional array, `unwrap` operates on the first nonsingleton dimension.

`Q = unwrap(P,tol)` uses a jump tolerance `tol` instead of the default value, π.

`Q = unwrap(P,[],dim)` unwraps along `dim` using the default tolerance.

`Q = unwrap(P,tol,dim)` uses a jump tolerance of `tol`.

Examples

Array `P` features smoothly increasing phase angles except for discontinuities at elements `(3,1)` and `(1,2)`.

```
P =
          0    7.0686    1.5708    2.3562
     0.1963    0.9817    1.7671    2.5525
     6.6759    1.1781    1.9635    2.7489
     0.5890    1.3744    2.1598    2.9452
```

The function `Q = unwrap(P)` eliminates these discontinuities.

```
Q =
          0    0.7854    1.5708    2.3562
     0.1963    0.9817    1.7671    2.5525
     0.3927    1.1781    1.9635    2.7489
     0.5890    1.3744    2.1598    2.9452
```

Limitations

The `unwrap` function detects branch cut crossings, but it can be fooled by sparse, rapidly changing phase values.

See Also

`abs`, `angle`

upper

Purpose Convert string to upper case

Syntax

```
t  = upper('str')
B = upper(A)
```

Description `t = upper('str')` converts any lower-case characters in the string *str* to the corresponding upper-case characters and leaves all other characters unchanged.

`B = upper(A)` when `A` is a cell array of strings, returns a cell array the same size as `A` containing the result of applying `upper` to each string within `A`.

Examples `upper('attention!')` is `ATTENTION!`.

Remarks Character sets supported:

- PC: Windows Latin-1
- Other: ISO Latin-1 (ISO 8859-1)

See Also `lower`

Purpose Determine if a Java feature is supported in MATLAB

Syntax

```
usejava(feature)
```

Description

`usejava(feature)` returns 1 if the specified feature is supported and 0 otherwise. Possible `feature` arguments are shown in the following table.

Feature	Description
'awt'	Abstract Window Toolkit components[1] are available
'desktop'	The MATLAB interactive desktop is running
'jvm'	The Java Virtual Machine is running
'swing'	Swing components[2] are available

1. Java's GUI components in the Abstract Window Tookit
2. Java's lightweight GUI components in the Java Foundation Classes

Examples

The following conditional code ensures that the AWT's GUI components are available before the M-file attempts to display a Java Frame.

```
if usejava('awt')
   myFrame = java.awt.Frame;
else
   disp('Unable to open a Java Frame');
end
```

The next example is part of an M-file that includes Java code. It fails gracefully when run in a MATLAB session that does not have access to a JVM.

```
if ~usejava('jvm')
   error([mfilename ' requires Java to run.']);
end
```

See Also

`javachk`

vander

Purpose Vandermonde matrix

Syntax

```
A = vander(v)
```

Description

`A = vander(v)` returns the Vandermonde matrix whose columns are powers of the vector `v`, that is, `A(i,j) = v(i)^(n-j)`, where `n = length(v)`.

Examples

```
vander(1:.5:3)

ans =

   1.0000    1.0000    1.0000    1.0000    1.0000
   5.0625    3.3750    2.2500    1.5000    1.0000
  16.0000    8.0000    4.0000    2.0000    1.0000
  39.0625   15.6250    6.2500    2.5000    1.0000
  81.0000   27.0000    9.0000    3.0000    1.0000
```

See Also `gallery`

Purpose Variance

Syntax

```
var(X)
var(X,1)
var(X,w)
```

Description `var(X)` returns the variance of `X` for vectors. For matrices, `var(X)`is a row vector containing the variance of each column of `X`. `var(X)` normalizes by N-1 where N is the sequence length. This makes `var(X)` the best unbiased estimate of the variance if `X` is a sample from a normal distribution.

`var(X,1)` normalizes by N and produces the second moment of the sample about its mean.

`var(X,W)` computes the variance using the weight vector `W`. The number of elements in `W` must equal the number of rows in `X` unless `W = 1`, which is treated as a short-cut for a vector of ones. The elements of `W` must be positive. `var` normalizes `W` by dividing each element in `W` by the sum of all its elements.

The variance is the square of the standard deviation (STD).

See Also `corrcoef`, `cov`, `std`

varargin, varargout

Purpose Pass or return variable numbers of arguments

Syntax

```
function varargout = foo(n)
function y = bar(varargin)
```

Description

`function varargout = foo(n)` returns a variable number of arguments from function `foo.m`.

`function y = bar(varargin)` accepts a variable number of arguments into function `bar.m`.

The `varargin` and `varargout` statements are used only inside a function M-file to contain the optional arguments to the function. Each must be declared as the last argument to a function, collecting all the inputs or outputs from that point onwards. In the declaration, `varargin` and `varargout` must be lowercase.

Examples

The function

```
function myplot(x,varargin)
plot(x,varargin{:})
```

collects all the inputs starting with the second input into the variable `varargin`. `myplot` uses the comma-separated list syntax `varargin{:}` to pass the optional parameters to `plot`. The call

```
myplot(sin(0:.1:1),'color',[.5 .7 .3],'linestyle',':')
```

results in `varargin` being a 1-by-4 cell array containing the values `'color'`, [.5 .7 .3], `'linestyle'`, and `':'`.

The function

```
function [s,varargout] = mysize(x)
nout = max(nargout,1)-1;
s = size(x);
for k=1:nout, varargout(k) = {s(k)}; end
```

returns the size vector and, optionally, individual sizes. So

```
[s,rows,cols] = mysize(rand(4,5));
```

returns `s = [4 5], rows = 4, cols = 5`.

See Also `nargin`, `nargout`, `nargchk`

vectorize

Purpose Vectorize expression

Syntax

```
vectorize(s)
vectorize(fun)
```

Description `vectorize(s)` where `s` is a string expression, inserts a `.` before any `^`, `*` or `/` in `s`. The result is a character string.

`vectorize(fun)` when `fun` is an inline function object, vectorizes the formula for `fun`. The result is the vectorized version of the inline function.

See Also `inline`, `cd`, `dbtype`, `delete`, `dir`, `partialpath`, `path`, `what`, `who`

Purpose Display version information for MATLAB, Simulink, and toolboxes

Graphical Interface As an alternative to the `ver` function, select **About** from the **Help** menu in any product that has a **Help** menu.

Syntax

```
ver
ver toolbox
v = ver('toolbox')
```

Description `ver` displays the current version numbers and release dates for MATLAB, Simulink, and all toolboxes.

`ver toolbox` displays the current version number and release date for the toolbox specified by `toolbox`. The name, `toolbox`, corresponds to the directory name that holds the `Contents.m` file for that toolbox. For example, `Contents.m` for the Fuzzy Logic Toolbox resides in the `fuzzy` directory. You therefore use `ver fuzzy` to obtain the version of this toolbox.

`v = ver('toolbox')` returns the version information in structure array, `v`, having fields `Name`, `Version`, `Release`, and `Date`.

Remarks See comments near the top of `ver.m` for information on how your own toolboxes can use the `ver` function. Type the following at the MATLAB command prompt.

```
type ver.m
```

Examples To return version information for the Fuzzy Logic Toolbox,

```
ver fuzzy
Fuzzy Logic Toolbox          Version 2.0.1 (R11)      16-Sep-1998
```

To return version information for MATLAB in a structure array, `v`,

```
v = ver('matlab')
v =
       Name: 'MATLAB Toolbox'
    Version: '6.0'
    Release: '(R12)'
       Date: '30-Dec-1999'
```

See Also

`help`, `version`, `whatsnew`

Also, type `help info` at the Command Window prompt.

Purpose Get MATLAB version number

Graphical Interface As an alternative to the `version` function, select **About** from the **Help** menu in the MATLAB desktop.

Syntax

```
version
version -java
v = version
[v,d] = version
```

Description `version` displays the MATLAB version number.

`version -java` displays the version of the Java VM used by MATLAB.

`v = version` returns a string `v` containing the MATLAB version number.

`[v,d] = version` also returns a string `d` containing the date of the version.

Examples

```
[v,d]=version

v =
6.0.0.60356 (R12)

d =
May  2 2000
```

See Also `help`, `ver`, `whatsnew`

Also, type `help info` at the Command Window prompt.

vertcat

Purpose

Vertical concatenation

Syntax

```
C = vertcat(A1,A2,...)
```

Description

`C = vertcat(A1,A2,...)` vertically concatenates matrices `A1`, `A2`, and so on. All matrices in the argument list must have the same number of columns.

`vertcat` concatenates N-dimensional arrays along the first dimension. The remaining dimensions must match.

MATLAB calls `C = vertcat(A1,A2,...)` for the syntax `C = [A1;A2;...]` when any of `A1`, `A2`, etc. is an object.

Examples

Create a 5-by-3 matrix, `A`, and a 3-by-3 matrix, `B`. Then vertically concatenate `A` and `B`.

```
A = magic(5);              % Create 5-by-3 matrix, A
A(:,4:5) = []

A =

    17    24     1
    23     5     7
     4     6    13
    10    12    19
    11    18    25


B = magic(3)*100           % Create 3-by-3 matrix, B

B =

   800   100   600
   300   500   700
   400   900   200


C = vertcat(A,B)           % Vertically concatenate A and B

C =
```

```
 17   24    1
 23    5    7
  4    6   13
 10   12   19
 11   18   25
800  100  600
300  500  700
400  900  200
```

See Also `horzcat`, `cat`

Purpose

Viewpoint specification

Syntax

```
view(az,el)
view([az,el])
view([x,y,z])
view(2)
view(3)
view(T)

[az,el] = view
T = view
```

Description

The position of the viewer (the viewpoint) determines the orientation of the axes. You specify the viewpoint in terms of azimuth and elevation, or by a point in three-dimensional space.

`view(az,el)` and `view([az,el])` set the viewing angle for a three-dimensional plot. The azimuth, `az`, is the horizontal rotation about the *z*-axis as measured in degrees from the negative *y*-axis. Positive values indicate counterclockwise rotation of the viewpoint. `el` is the vertical elevation of the viewpoint in degrees. Positive values of elevation correspond to moving above the object; negative values correspond to moving below the object.

`view([x,y,z])` sets the viewpoint to the Cartesian coordinates `x`, `y`, and `z`. The magnitude of `(x,y,z)` is ignored.

`view(2)` sets the default two-dimensional view, `az = 0, el = 90`.

`view(3)` sets the default three-dimensional view, `az = –37.5, el = 30`.

`view(T)` sets the view according to the transformation matrix `T`, which is a 4-by-4 matrix such as a perspective transformation generated by `viewmtx`.

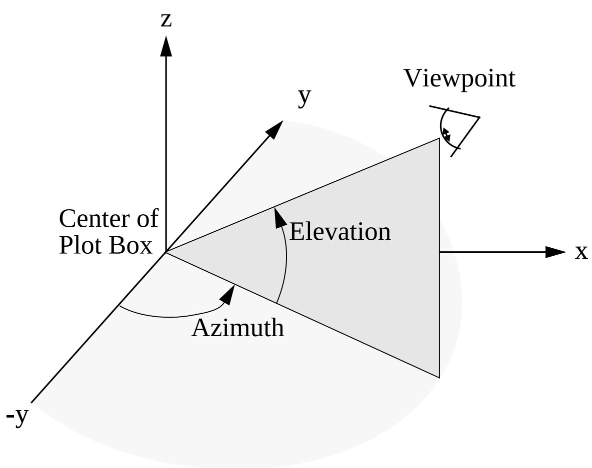
`[az,el] = view` returns the current azimuth and elevation.

`T = view` returns the current 4-by-4 transformation matrix.

Remarks

Azimuth is a polar angle in the *x*-*y* plane, with positive angles indicating counterclockwise rotation of the viewpoint. Elevation is the angle above (positive angle) or below (negative angle) the *x*-*y* plane.

This diagram illustrates the coordinate system. The arrows indicate positive directions.



Examples

View the object from directly overhead.

```
az = 0;
el = 90;
view(az, el);
```

Set the view along the *y*-axis, with the *x*-axis extending horizontally and the *z*-axis extending vertically in the figure.

```
view([0 0]);
```

Rotate the view about the *z*-axis by 180°.

```
az = 180;
el = 90;
view(az, el);
```

See Also

`viewmtx`, `axes`, `rotate3d`

axes graphics object properties: `CameraPosition`, `CameraTarget`, `CameraViewAngle`, `Projection`.

Purpose View transformation matrices

Syntax

```
T = viewmtx(az,el)
T = viewmtx(az,el,phi)
T = viewmtx(az,el,phi,xc)
```

Description

`viewmtx` computes a 4-by-4 orthographic or perspective transformation matrix that projects four-dimensional homogeneous vectors onto a two-dimensional view surface (e.g., your computer screen).

`T = viewmtx(az,el)` returns an *orthographic* transformation matrix corresponding to azimuth `az` and elevation `el`. `az` is the azimuth (i.e., horizontal rotation) of the viewpoint in degrees. `el` is the elevation of the viewpoint in degrees. This returns the same matrix as the commands

```
view(az,el)
T = view
```

but does not change the current view.

`T = viewmtx(az,el,phi)` returns a *perspective* transformation matrix. `phi` is the perspective viewing angle in degrees. `phi` is the subtended view angle of the normalized plot cube (in degrees) and controls the amount of perspective distortion.

Phi	Description
0 degrees	Orthographic projection
10 degrees	Similar to telephoto lens
25 degrees	Similar to normal lens
60 degrees	Similar to wide angle lens

You can use the matrix returned to set the view transformation with `view(T)`. The 4-by-4 perspective transformation matrix transforms four-dimensional homogeneous vectors into unnormalized vectors of the form (x,y,z,w), where w is not equal to 1. The x- and y-components of the normalized vector (x/w, y/w, z/w, 1) are the desired two-dimensional components (see example below).

T = viewmtx(az,el,phi,xc) returns the perspective transformation matrix using xc as the target point within the normalized plot cube (i.e., the camera is looking at the point xc). xc is the target point that is the center of the view. You specify the point as a three-element vector, xc = [xc,yc,zc], in the interval [0,1]. The default value is xc = [0,0,0].

Remarks

A four-dimensional homogenous vector is formed by appending a 1 to the corresponding three-dimensional vector. For example, [x,y,z,1] is the four-dimensional vector corresponding to the three-dimensional point [x,y,z].

Examples

Determine the projected two-dimensional vector corresponding to the three-dimensional point (0.5,0.0,-3.0) using the default view direction. Note that the point is a column vector.

```
A = viewmtx(-37.5,30);
x4d = [.5  0  -3  1]';
x2d = A*x4d;
x2d = x2d(1:2)
x2d =
     0.3967
    -2.4459
```

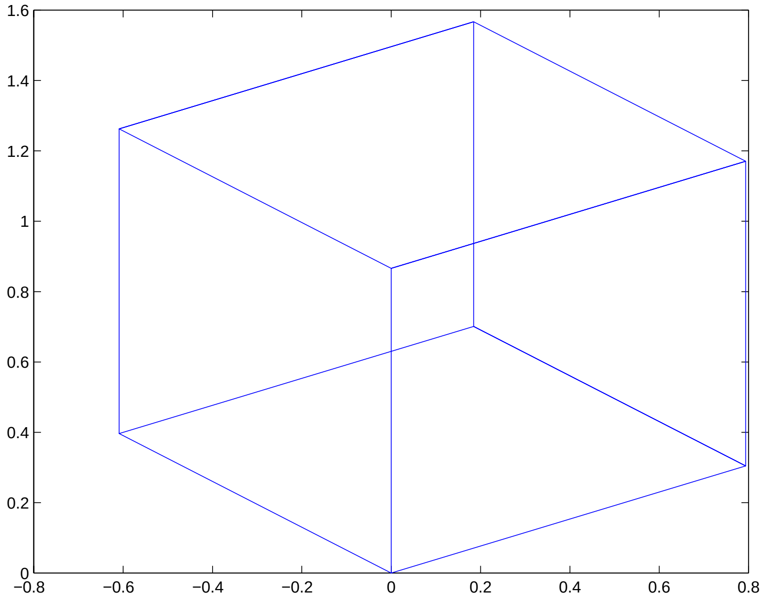
Vectors that trace the edges of a unit cube are

```
x = [0  1  1  0  0  0  1  1  0  0  1  1  1  1  0  0];
y = [0  0  1  1  0  0  0  1  1  0  0  0  1  1  1  1];
z = [0  0  0  0  0  1  1  1  1  1  1  0  0  1  1  0];
```

Transform the points in these vectors to the screen, then plot the object.

```
A = viewmtx(-37.5,30);
[m,n] = size(x);
x4d = [x(:),y(:),z(:),ones(m*n,1)]';
x2d = A*x4d;
x2 = zeros(m,n); y2 = zeros(m,n);
x2(:) = x2d(1,:);
y2(:) = x2d(2,:);
```

```
plot(x2,y2)
```



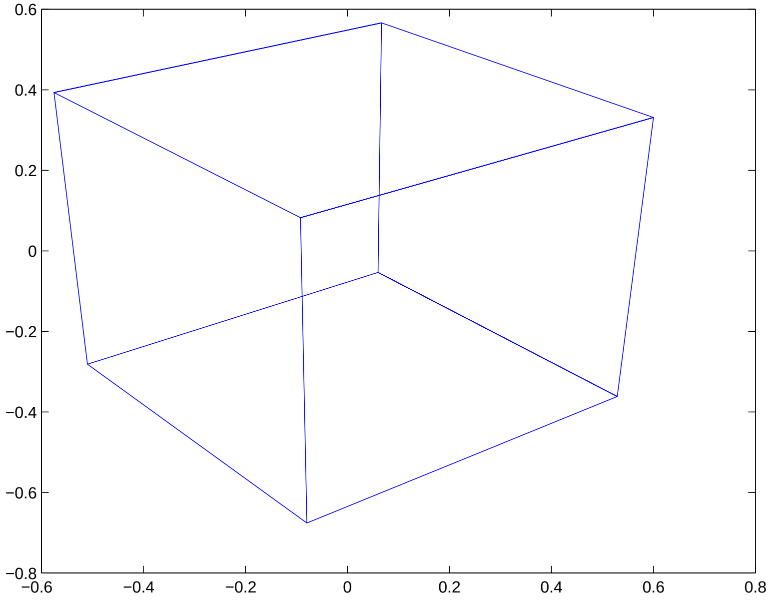
Use a perspective transformation with a 25 degree viewing angle:

```
A = viewmtx(-37.5,30,25);
x4d = [.5  0  -3  1]';
x2d = A*x4d;
x2d = x2d(1:2)/x2d(4)   % Normalize
x2d =
     0.1777
    -1.8858
```

Transform the cube vectors to the screen and plot the object:

```
A = viewmtx(-37.5,30,25);
[m,n] = size(x);
x4d = [x(:),y(:),z(:),ones(m*n,1)]';
x2d = A*x4d;
x2 = zeros(m,n); y2 = zeros(m,n);
x2(:) = x2d(1,:)./x2d(4,:);
y2(:) = x2d(2,:)./x2d(4,:);
```

```
plot(x2,y2)
```



See Also

`view`

Purpose

Returns coordinate and color limits for volume data

Syntax

```
lims = volumebounds(X,Y,Z,V)
lims = volumebounds(X,Y,Z,U,V,W)
lims = volumebounds(V), lims = volumebounds(U,V,W)
```

Description

`lims = volumebounds(X,Y,Z,V)` returns the x,y,z and color limits of the current axes for scalar data. `lims` is returned as a vector:

```
[xmin xmax ymin ymax zmin zmax cmin cmax]
```

You can pass this vector to the `axis` command.

`lims = volumebounds(X,Y,Z,U,V,W)` returns the x, y, and z limits of the current axes for vector data. `lims` is returned as a vector:

```
[xmin xmax ymin ymax zmin zmax]
```

`lims = volumebounds(V), lims = volumebounds(U,V,W)` assumes `X`, `Y`, and `Z` are determined by the expression:

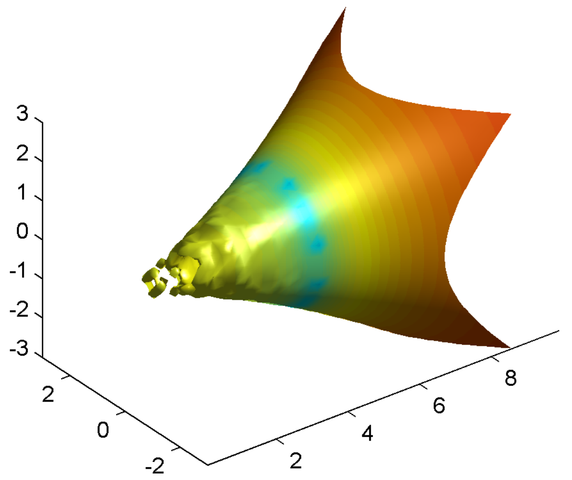
```
[X Y Z] = meshgrid(1:n,1:m,1:p)
```

where `[m n p] = size(V)`.

Examples

This example uses `volumebounds` to set the axis and color limits for an `isosurface` generated by the `flow` function.

```
[x y z v] = flow;
p = patch(isosurface(x,y,z,v,-3));
isonormals(x,y,z,v,p)
daspect([1 1 1])
isocolors(x,y,z,flipdim(v,2),p)
shading interp
axis(volumebounds(x,y,z,v))
view(3)
camlight
lighting phong
```



See Also isosurface, streamslice

Purpose Voronoi diagram

Syntax

```
voronoi(x,y)
voronoi(x,y,TRI)
voronoi(...,'LineSpec')
h = voronoi(...)
[vx,vy] = voronoi(...)
```

Definition Consider a set of coplanar points P. For each point P_x in the set P, you can draw a boundary enclosing all the intermediate points lying closer to P_x than to other points in the set P. Such a boundary is called a *Voronoi polygon*, and the set of all Voronoi polygons for a given point set is called a *Voronoi diagram*.

Description `voronoi(x,y)` plots the bounded cells of the Voronoi diagram for the points `x`,`y`. Cells that contain a point at infinity are unbounded and are not plotted.

`voronoi(x,y,TRI)` uses the triangulation `TRI` instead of computing it via `delaunay`.

`voronoi(...,'LineSpec')` plots the diagram with color and line style specified.

`h = voronoi(...)` returns, in `h`, handles to the line objects created.

`[vx,vy] = voronoi(...)` returns the finite vertices of the Voronoi edges in `vx` and `vy` so that `plot(vx,vy,'-',x,y,'.')` creates the Voronoi diagram.

Note For the topology of the Voronoi diagram, i.e., the vertices for each Voronoi cell, use `voronoin`.

```
[v,c] = voronoin([x(:) y(:)])
```

Visualization Use one of these methods to plot a Voronoi diagram:

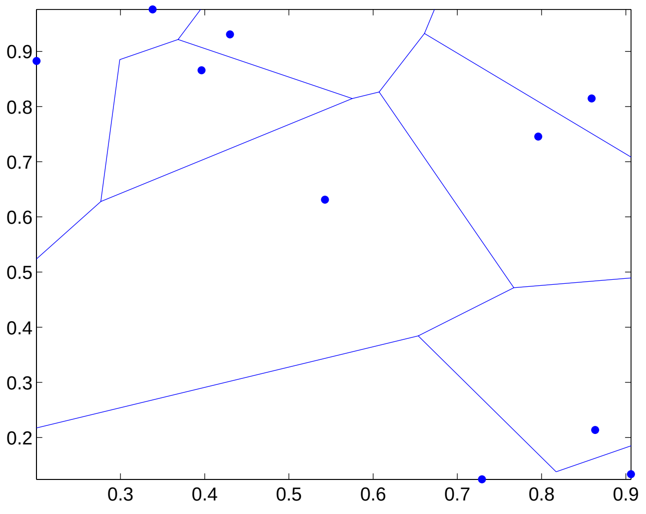
- If you provide no output argument, `voronoi` plots the diagram. See Example 1.

- To gain more control over color, line style, and other figure properties, use the syntax `[vx,vy] = voronoi(...)`. This syntax returns the vertices of the finite Voronoi edges, which you can then plot with the `plot` function. See Example 2.
- To fill the cells with color, use `voronoin` with `n = 2` to get the indices of each cell, and then use `patch` and other plot functions to generate the figure. Note that `patch` does not fill unbounded cells with color. See Example 3.

Examples

Example 1. This code uses the `voronoi` function to plot the Voronoi diagram for 10 randomly generated points.

```
rand('state',5);
x = rand(1,10); y = rand(1,10);
voronoi(x,y)
```



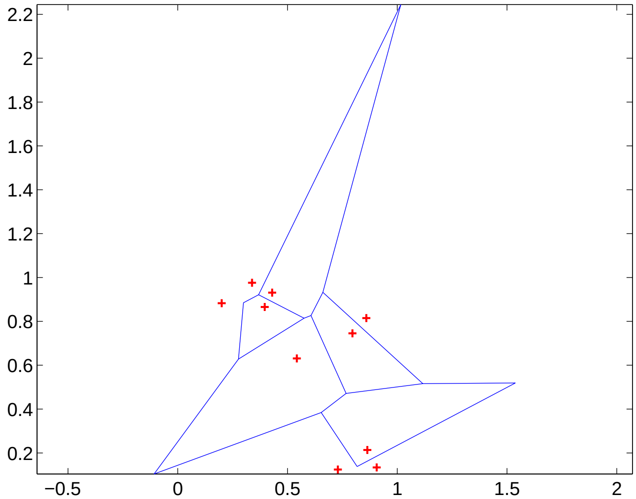
Example 2. This code uses the vertices of the finiteVoronoi edges to plot the Voronoi diagram for the same 10 points.

```
rand('state',5);
x = rand(1,10); y = rand(1,10);
[vx, vy] = voronoi(x,y);
```

```
plot(x,y,'r+',vx,vy,'b-'); axis equal
```

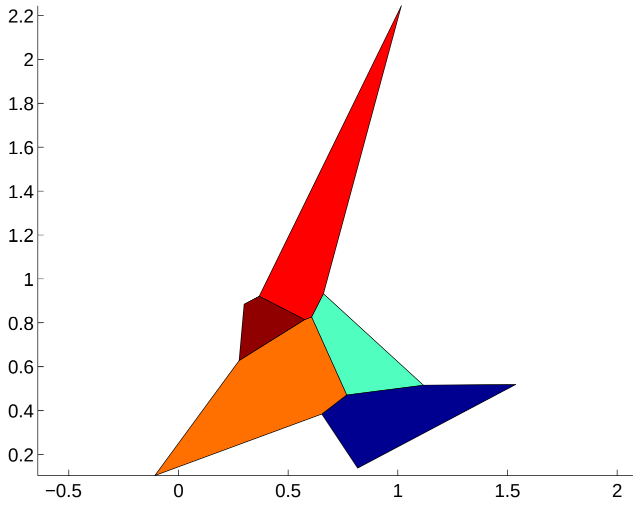
Note that you can add this code to get the figure shown in Example 1.

```
xlim([min(x) max(x)])
ylim([min(y) max(y)])
```



Example 3. This code uses voronoin and patch to fill the bounded cells of the same Voronoi diagram with color.

```
rand('state',5);
x=rand(10,2);
[v,c]=voronoin(x);
for i = 1:length(c)
if all(c{i}~=1)   % If at least one of the indices is 1,
                  % then it is an open region and we can't
                  % patch that.
patch(v(c{i},1),v(c{i},2),i); % use color i.
end
end
axis equal
```



See Also `convhull`, `delaunay`, `LineSpec`, `plot`, `voronoin`

Purpose n-D Voronoi diagram

Syntax

```
[V,C] = voronoin(X)
```

Description `[V,C] = voronoin(X)` returns Voronoi vertices `V` and the Voronoi cells `C` of the Voronoi diagram of `X`. `V` is a `numv`-by-`n` array of the `numv` Voronoi vertices in `n`-D space, each row corresponds to a Voronoi vertex. `C` is a vector cell array where each element contains the indices into `V` of the vertices of the corresponding Voronoi cell. `X` is an `m`-by-`n` array, representing `m` `n`-D points, where `n > 1` and `m >= n+1`.

The first row of `V` is a point at infinity. If any index in a cell of the cell array is `1`, then the corresponding Voronoi cell contains the first point in `V`, a point at infinity. This means the Voronoi cell is unbounded.

Note `voronoin` is based on `qhull` [2]. For information about `qhull`, see `http://www.geom.umn.edu/software/qhull/`. For copyright information, see `http://www.geom.umn.edu/software/download/COPYING.html`.

Visualization You can plot individual bounded cells of an `n`-D Voronoi diagram. To do this, use `convhulln` to compute the vertices of the facets that make up the Voronoi cell. Then use `patch` and other plot functions to generate the figure. For an example, see "Tessellation and Interpolation of Scattered Data in Higher Dimensions" in the MATLAB documentation.

Examples Let

```
x = [ 0.5    0
      0      0.5
     -0.5   -0.5
     -0.2   -0.1
     -0.1    0.1
      0.1   -0.1
      0.1    0.1 ]
```

then

```
[V,C] = voronoin(x)
```

```
V =
       Inf       Inf
    0.3833    0.3833
    0.7000   -1.6500
    0.2875    0.0000
   -0.0000    0.2875
   -0.0000   -0.0000
   -0.0500   -0.5250
   -0.0500   -0.0500
   -1.7500    0.7500
   -1.4500    0.6500

C =

    [1x4 double]
    [1x5 double]
    [1x4 double]
    [1x4 double]
    [1x4 double]
    [1x5 double]
    [1x4 double]
```

Use a `for` loop to see the contents of the cell array `C`.

```
for i=1:length(C), disp(C{i}), end

     4     2     1     3
    10     5     2     1     9
     9     1     3     7
    10     8     7     9
    10     5     6     8
     8     6     4     3     7
     6     4     2     5
```

In particular, the fifth Voronoi cell consists of 4 points: `V(10,:)`, `V(5,:)`, `V(6,:)`, `V(8,:)`.

See Also

`convhull`, `convhulln`, `delaunay`, `delaunayn`, `voronoi`

Reference

[1] Barber, C. B., D.P. Dobkin, and H.T. Huhdanpaa, "The Quickhull Algorithm for Convex Hulls," *ACM Transactions on Mathematical Software*, Vol. 22, No. 4, Dec. 1996, p. 469-483. Available in HTML format at `http://www.acm.org/pubs/citations/journals/toms/1996-22-4/p469-barber/` and in PostScript format at `ftp://geom.umn.edu/pub/software/qhull-96.ps`.

[2] National Science and Technology Research Center for Computation and Visualization of Geometric Structures (The Geometry Center), University of Minnesota. 1993.

waitbar

Purpose

Display waitbar

Syntax

```
h = waitbar(x,'title')
waitbar(x,'title','CreateCancelBtn','button_callback')
waitbar(...,property_name,property_value,...)
waitbar(x)
waitbar(x,h)
waitbar(x,h,'updated title')
```

Description

A waitbar shows what percentage of a calculation is complete, as the calculation proceeds.

`h = waitbar(x,'title')` displays a waitbar of fractional length `x`. The handle to the waitbar figure is returned in `h`. `x` must be between 0 and 1.

`waitbar(x,'title','`**`CreateCancelBtn`**`','button_callback')` specifying **`CreateCancelBtn`** adds a cancel button to the figure that executes the MATLAB commands specified in `button_callback` when the user clicks the cancel button or the close figure button. `waitbar` sets both the cancel button callback and the figure `CloseRequestFcn` to the string specified in `button_callback`.

`waitbar(...,property_name,property_value,...)` optional arguments `property_name` and `property_value` enable you to set corresponding `waitbar` figure properties.

`waitbar(x)` subsequent calls to `waitbar(x)` extend the length of the bar to the new position `x`.

`waitbar(x,h)` extends the length of the bar in the waitbar `h` to the new position `x`.

Example

`waitbar` is typically used inside a `for` loop that performs a lengthy computation. For example,

```
h = waitbar(0,'Please wait...');

for i=1:100, % computation here %
waitbar(i/100)
end
```

```
close(h)
```

waitfor

Purpose

Wait for condition

Syntax

```
waitfor(h)
waitfor(h,'PropertyName')
waitfor(h,'PropertyName',PropertyValue)
```

Description

The `waitfor` function blocks the caller's execution stream so that command-line expressions, callbacks, and statements in the blocked M-file do not execute until a specified condition is satisfied.

`waitfor(h)` returns when the graphics object identified by `h` is deleted or when a **Ctrl-C** is typed in the Command Window. If `h` does not exist, `waitfor` returns immediately without processing any events.

`waitfor(h,'PropertyName')`, in addition to the conditions in the previous syntax, returns when the value of `'PropertyName'` for the graphics object `h` changes. If `'PropertyName'` is not a valid property for the object, `waitfor` returns immediately without processing any events.

`waitfor(h,'PropertyName',PropertyValue)`, in addition to the conditions in the previous syntax, `waitfor` returns when the value of `'PropertyName'` for the graphics object `h` changes to `PropertyValue`. `waitfor` returns immediately without processing any events if `'PropertyName'` is set to `PropertyValue`.

Remarks

While `waitfor` blocks an execution stream, other execution streams in the form of callbacks may execute as a result of various events (e.g., pressing a mouse button).

`waitfor` can block nested execution streams. For example, a callback invoked during a `waitfor` statement can itself invoke `waitfor`.

See Also

`uiresume`, `uiwait`

Purpose

Wait for key or mouse button press

Syntax

`k = waitforbuttonpress`

Description

`k = waitforbuttonpress` blocks the caller's execution stream until the function detects that the user has pressed a mouse button or a key while the figure window is active. The function returns

- `0` if it detects a mouse button press
- `1` if it detects a key press

Additional information about the event that causes execution to resume is available through the figure's `CurrentCharacter`, `SelectionType`, and `CurrentPoint` properties.

If a `WindowButtonDownFcn` is defined for the figure, its callback is executed before `waitforbuttonpress` returns a value.

Example

These statements display text in the Command Window when the user either clicks a mouse button or types a key in the figure window:

```
w = waitforbuttonpress;
if w == 0
    disp('Button press')
else
    disp('Key press')
end
```

See Also

`dragrect`, `figure`, `gcf`, `ginput`, `rbbox`, `waitfor`

warndlg

Purpose Display warning dialog box

Syntax

```
h = warndlg('warningstring','dlgname')
```

Description

`warndlg` displays a dialog box named `'Warning Dialog'` containing the string `'This is the default warning string.'` The warning dialog box disappears after you press the **OK** button.

`warndlg('warningstring')` displays a dialog box with the title `'Warning Dialog'` containing the string specified by `warningstring`.

`warndlg('warningstring','dlgname')` displays a dialog box with the title `dlgname` that contains the string `warningstring`.

`h = warndlg(...)` returns the handle of the dialog box.

Examples

The statement

```
warndlg('Pressing OK will clear memory','!! Warning !!')
```

displays this dialog box:

See Also

`dialog`, `errordlg`, `helpdlg`, `msgbox`

Purpose Display warning message

Syntax

```
warning('message')
warning on
warning off
warning backtrace
warning debug
warning once
warning always
[s,f] = warning
```

Description

`warning('message')` displays the text `'message'` as does the `disp` function, except that with `warning`, message display can be suppressed.

`warning off` suppresses all subsequent warning messages.

`warning on` re-enables them.

`warning backtrace` is the same as `warning on` except that the file and line number that produced the warning are displayed.

`warning debug` is the same as `dbstop if warning` and triggers the debugger when a warning is encountered.

`warning once` displays Handle Graphics backwards compatibility warnings only once per session.

`warning always` displays Handle Graphics backwards compatibility warnings as they are encountered (subject to current warning state).

`[s,f] = warning` returns the current warning state `s` and the current warning frequency `f` as strings.

Remarks Use `dbstop on warning` to trigger the debugger when a warning is encountered.

See Also `dbstop`, `disp`, `error`, `errordlg`

waterfall

Purpose

Waterfall plot

Syntax

```
waterfall(Z)
waterfall(X,Y,Z)
waterfall(...,C)

h = waterfall(...)
```

Description

The `waterfall` function draws a mesh similar to the `meshz` function, but it does not generate lines from the columns of the matrices. This produces a "waterfall" effect.

`waterfall(Z)` creates a waterfall plot using `x = 1:size(Z,1)` and `y = 1:size(Z,1)`. `Z` determines the color, so color is proportional to surface height.

`waterfall(X,Y,Z)` creates a waterfall plot using the values specified in `X`, `Y`, and `Z`. `Z` also determines the color, so color is proportional to the surface height. If `X` and `Y` are vectors, `X` corresponds to the columns of `Z`, and `Y` corresponds to the rows, where `length(x) = n`, `length(y) = m`, and `[m,n] = size(Z)`. `X` and `Y` are vectors or matrices that define the *x* and *y* coordinates of the plot. `Z` is a matrix that defines the *z* coordinates of the plot (i.e., height above a plane). If `C` is omitted, color is proportional to `Z`.

`waterfall(...,C)` uses scaled color values to obtain colors from the current colormap. Color scaling is determined by the range of `C`, which must be the same size as `Z`. MATLAB performs a linear transformation on `C` to obtain colors from the current colormap.

`h = waterfall(...)` returns the handle of the patch graphics object used to draw the plot.

Remarks

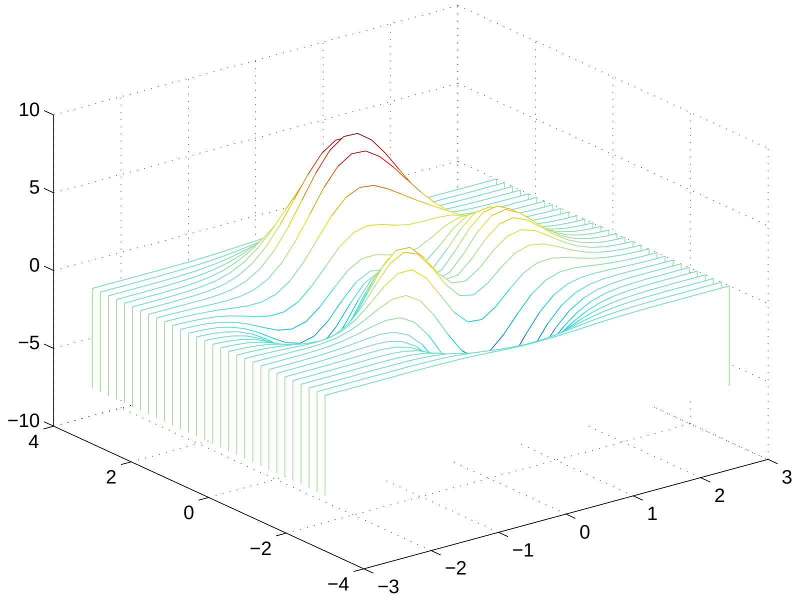
For column-oriented data analysis, use `waterfall(Z')` or `waterfall(X',Y',Z')`.

Examples

Produce a waterfall plot of the peaks function.

```
[X,Y,Z] = peaks(30);
```

```
waterfall(X,Y,Z)
```



Algorithm The range of X, Y, and Z, or the current setting of the axes Llim, YLim, and ZLim properties, determines the range of the axes (also set by axis). The range of C, or the current setting of the axes Clim property, determines the color scaling (also set by caxis).

The CData property for the patch graphics objects specifies the color at every point along the edge of the patch, which determines the color of the lines.

The waterfall plot looks like a mesh surface; however, it is a patch graphics object. To create a surface plot similar to waterfall, use the meshz function and set the MeshStyle property of the surface to 'Row'. For a discussion of parametric surfaces and related color properties, see surf.

See Also axes, axis, caxis, meshz, ribbon, surf

Properties for patch graphics objects.

wavplay

Purpose

Play recorded sound on a PC-based audio output device.

Syntax

```
wavplay(y,Fs)
wavplay(...,'mode')
```

Description

`wavplay(y,Fs)` plays the audio signal stored in the vector `y` on a PC-based audio output device. You specify the audio signal sampling rate with the integer `Fs` in samples per second. The default value for `Fs` is 11025 Hz (samples per second).

`wavplay(...,'mode')` specifies how `wavplay` interacts with the command line, according the string `'mode'`. The string `'mode'` can be:

- `'async'` (default value): You have immediate access to the command line as soon as the sound begins to play on the audio output device (a nonblocking device call).
- `'sync'`: You don't have access to the command line until the sound has finished playing (a blocking device call).

The audio signal `y` can be one of four data types. The number of bits used to quantize and play back each sample depends on the data type.

Table 2-1: Data Types for wavplay

Data Type	Quantization
Double-precision (default value)	16 bits/sample
Single-precision	16 bits/sample
16-bit signed integer	16 bits/sample
8-bit unsigned integer	8 bits/sample

Remarks

You can play your signal in stereo if `y` is a two-column matrix.

Examples

The MAT-files `gong.mat` and `chirp.mat` both contain an audio signal `y`, and a sampling frequency `Fs`. Load and play the gong and the chirp audio signals. Change the names of these signals in between `load` commands and play them sequentially using the `'sync'` option for `wavplay`.

```
load chirp;
y1 = y; Fs1 = Fs;
load gong;
wavplay(y1,Fs1,'sync') % The chirp signal finishes before the
wavplay(y,Fs)          % gong signal begins playing.
```

See Also wavrecord

wavread

Purpose

Read Microsoft WAVE (`.wav`) sound file

Graphical Interface

As an alternative to `auread`, use the Import Wizard. To activate the Import Wizard, select **Import Data** from the **File** menu.

Syntax

```
y = wavread('filename')
[y,Fs,bits] = wavread('filename')
[...] = wavread('filename',N)
[...] = wavread('filename',[N1 N2])
[...] = wavread('filename','size')
```

Description

`wavread` supports multichannel data, with up to 16 bits per sample.

`y = wavread('filename')` loads a WAVE file specified by the string `filename`, returning the sampled data in `y`. The `.wav` extension is appended if no extension is given. Amplitude values are in the range `[-1,+1]`.

`[y,Fs,bits] = wavread('filename')` returns the sample rate (`Fs`) in Hertz and the number of bits per sample (`bits`) used to encode the data in the file.

`[...] = wavread('filename',N)` returns only the first `N` samples from each channel in the file.

`[...] = wavread('filename',[N1 N2])` returns only samples `N1` through `N2` from each channel in the file.

`siz = wavread('filename','size')` returns the size of the audio data contained in the file in place of the actual audio data, returning the vector `siz = [samples channels]`.

See Also

`auread`, `wavwrite`

Purpose Record sound using a PC-based audio input device.

Syntax

```
y = wavrecord(n,Fs)
y = wavrecord(...,ch)
y = wavrecord(...,'dtype')
```

Description `y = wavrecord(n,Fs)` records `n` samples of an audio signal, sampled at a rate of `Fs` Hz (samples per second). The default value for `Fs` is 11025 Hz.

`y = wavrecord(...,ch)` uses `ch` number of input channels from the audio device. The default value for `ch` is 1.

`y = wavrecord(...,'dtype')` uses the data type specified by the string `'dtype'` to record the sound. The string `'dtype'` can be one of the following:

- `'double'` (default value), 16 bits/sample
- `'single'`, 16 bits/sample
- `'int16'`, 16 bits/sample
- `'uint8'`, 8 bits/sample

Remarks Standard sampling rates for PC-based audio hardware are 8000, 11025, 2250, and 44100 samples per second. Stereo signals are returned as two-column matrices. The first column of a stereo audio matrix corresponds to the left input channel, while the second column corresponds to the right input channel.

Examples Record 5 seconds of 16-bit audio sampled at 11,025 Hz. Play back the recorded sound using `wavplay`. Speak into your audio device (or produce your audio signal) while the `wavrecord` command runs.

```
Fs = 11025;
y = wavrecord(5*Fs,Fs,'int16');
wavplay(y,Fs);
```

See Also `wavplay`

wavwrite

Purpose Write Microsoft WAVE (`.wav`) sound file

Syntax

```
wavwrite(y,'filename')
wavwrite(y,Fs,'filename')
wavwrite(y,Fs,N,'filename')
```

Description `wavwrite` supports multi-channel 8- or 16-bit WAVE data.

`wavwrite(y,'filename')` writes a WAVE file specified by the string `filename`. The data should be arranged with one channel per column. Amplitude values outside the range `[-1,+1]` are clipped prior to writing.

`wavwrite(y,Fs,'filename')` specifies the sample rate `Fs`, in Hertz, of the data.

`wavwrite(y,Fs,N,'filename')` forces an `N`-bit file format to be written, where `N <= 16`.

See Also `auwrite`, `wavread`

Purpose Point Help browser or Web browser to file or Web site

Graphical Interface As an alternative to the `web` function, type the URL in the page title field at the top of the display pane in the Help browser.

Syntax

```
web url
web url -browser
stat = web('url', '-browser')
```

Description

`web url` displays the MATLAB Help browser, loads the file or Web site specified by `url` (the URL) in it, and returns the status to the Command Window. Generally, `url` specifies a local file or a Web site on the Internet.

`web url -browser` displays the default Web browser for your system, loads the file or Web site specified by `url` (the URL) in it, and returns the status to the Command Window. Generally, `url` specifies a local file or a Web site on the Internet. The URL can be in any form that the browser supports. On Windows, the default Web browser is determined by the operating system. On UNIX, the Web browser used is specified in `docopt`, in the `doccmd` string. If your system default browser is Netscape, start Netscape before issuing the `web` function with the `-browser` argument to avoid possible problems.

`stat = web('url', '-browser')` is the function form and returns the status of `web` to the variable `stat`.

Value of stat	Description
0	Browser was found and launched.
1	Browser was not found.
2	Browser was found but could not be launched.

Examples

`web file:/disk/dir1/dir2/foo.html` points the Help browser to the file `foo.html`. If the file is on the MATLAB path, `web(['file:' which('foo.html')])` also works.

`web http://www.mathworks.com` loads The MathWorks Web page into the Help browser.

`web www.mathworks.com -browser` loads The MathWorks Web page into your system's default Web browser, for example, Netscape Navigator.

Use `web mailto:email_address` to use your default e-mail application to send a message to `email_address`.

See Also

`doc`, `docopt`, `helpbrowser`

Purpose Day of the week

Syntax

```
[N,S] = weekday(D)
```

Description `[N,S] = weekday(D)` returns the day of the week in numeric (`N`) and string (`S`) form for each element of a serial date number array or date string. The days of the week are assigned these numbers and abbreviations:

N	S	N	S
1	Sun	5	Thu
2	Mon	6	Fri
3	Tue	7	Sat
4	Wed		

Examples Either

```
[n,s] = weekday(728647)
```

or

```
[n,s] = weekday('19-Dec-1994')
```

returns `n = 2` and `s = Mon`.

See Also `datenum`, `datevec`, `eomday`

what

Purpose

List MATLAB specific files in current directory

Graphical Interface

As an alternative to the what function, use the Current Directory browser. To open it, select **Current Directory** from the **View** menu in the MATLAB desktop.

Syntax

```
what
what dirname
s = what('dirname')
```

Description

`what` lists the `M`, `MAT`, `MEX`, `MDL`, and `P`-files and the class directories that reside in the current working directory.

`what dirname` lists the files in directory `dirname` on the MATLAB search path. It is not necessary to enter the full pathname of the directory. The last component, or last couple of components, is sufficient.

Use `what class` to list the files in method directory, `@class`. For example, `what cfit` lists the MATLAB files in `toolbox\curvefit\curvefit\@cfit`.

`s = what('dirname')` returns the results in a structure array with these fields.

Field	Description
`path`	Path to directory
`m`	Cell array of M-file names
`mat`	Cell array of MAT-file names
`mex`	Cell array of MEX-file names
`mdl`	Cell array of MDL-file names
`p`	Cell array of P-file names
`classes`	Cell array of class names

`what dirname` is the unquoted form of the syntax.

Examples

To list the files in `toolbox\matlab\audio`,

```
what audio

M-files in directory matlabroot\toolbox\matlab\audio

Contents    lin2mu      sound       wavread
auread      mu2lin      soundsc     wavrecord
auwrite     saxis       wavplay     wavwrite

MAT-files in directory matlabroot\toolbox\matlab\audio

chirp        handel       splat
gong         laughter     train
```

To obtain a structure array containing the MATLAB filenames in `toolbox\matlab\general`, type

```
s = what('general')
s =
       path: 'matlabroot:\toolbox\matlab\general'
          m: {105x1 cell}
        mat: {0x1 cell}
        mex: {5x1 cell}
        mdl: {0x1 cell}
          p: {'helpwin.p'}
    classes: {'char'}
```

See Also

`dir`, `exist`, `lookfor`, `path`, `which`, `who`

whatsnew

Purpose Display README files for MATLAB and toolboxes

Syntax

```
whatsnew
whatsnew matlab
whatsnew toolboxpath
```

Description `whatsnew` displays the `README` file for the MATLAB product or a specified toolbox. If present, the `README` file summarizes new functionality that is not described in the documentation.

`whatsnew` **`matlab`** displays the `README` file for MATLAB.

`whatsnew toolboxpath` displays the `README` file for the toolbox specified by the string `toolboxpath`.

Examples To display the `README` file for MATLAB, type

```
whatsnew matlab
```

To display the `README` file for the Signal Processing Toolbox, type

```
whatsnew signal
```

See Also `help`, `lookfor`, `path`, `version`, `which`

Purpose

Locate functions and files

Graphical Interface

As an alternative to the `which` function, use the Current Directory browser. To open it, select **Current Directory** from the **View** menu in the MATLAB desktop.

Syntax

```
which fun
which classname/fun
which private/fun
which classname/private/fun
which fun1 in fun2
which fun(a,b,c,...)
which file.ext
which fun -all
s = which('fun',...)
```

Description

`which fun` displays the full pathname for the argument `fun`. If `fun` is a

- MATLAB function or Simulink model in an M, P, or MDL file on the MATLAB path, then `which` displays the full pathname for the corresponding file
- Workspace variable or built-in function, then `which` displays a message identifying `fun` as a variable or built-in function
- Method in a loaded Java class, then `which` displays the package, class, and method name for that method

If `fun` is an overloaded function or method, then `which fun` returns only the pathname of the first function or method found.

`which classname/fun` displays the full pathname for the M-file defining the `fun` method in MATLAB class, `classname`. For example, `which serial/fopen` displays the path for `fopen.m` in MATLAB class directory, `@serial`.

`which private/fun` limits the search to private functions. For example, `which private/orthog` displays the path for `orthog.m` in the `\private` subdirectory of `toolbox\matlab\elmat`.

`which classname/private/fun` limits the search to private methods defined by the MATLAB class, `classname`. For example, `which dfilt/private/todtf` displays the path for `todtf.m` in the `private` directory of the `dfilt` class.

`which fun1 in fun2` displays the pathname to function `fun1` in the context of the M-file `fun2`. You can use this form to determine whether a subfunction or private version of `fun1` is called from `fun2`, rather than a function on the path. For example, `which get in editpath` tells you which `get` function is called by `editpath.m`.

During debugging of `fun2`, using `which fun1` gives the same result.

`which fun(a,b,c,...)` displays the path to the specified function with the given input arguments. For example, if `d` is a database driver object, then `which get(d)` displays the path `toolbox\database\database\@driver\get.m`.

`which file.ext` displays the full pathname of the specified file if that file is in the current working directory or on the MATLAB path. Use `exist` to check for existence of files anywhere else.

`which fun -all` displays the paths to all items on the MATLAB path with the name `fun`. The first item in the returned list is usually the one that would be returned by `which` without using **`-all`**. The others in the list either are shadowed or can be executed in special circumstances. You may use the **`-all`** qualifier with any of the above formats of the `which` function.

`s = which('fun',...)` returns the results of `which` in the string `s`. For built-in functions or workspace variables, `s` will be the string `built-in` or `variable`, respectively. You may specify an output variable in any of the above formats of the `which` function.

If **`-all`** is used with this form, the output `s` is always a cell array of strings, even if only one string is returned.

Examples

The first statement below reveals that `inv` is a built-in function. The second indicates that `pinv` is in the `matfun` directory of the MATLAB Toolbox.

```
which inv
inv is a built-in function.

which pinv
```

```
matlabroot\toolbox\matlab\matfun\pinv.m
```

To find the fopen function used on MATLAB serial class objects

```
which serial/fopen
matlabroot\toolbox\matlab\iofun\@serial\fopen.m  % serial method
```

To find the setTitle method used on objects of the Java Frame class, the class must first be loaded into MATLAB. The class is loaded when you create an instance of the class.

```
frameObj = java.awt.Frame;

which setTitle
java.awt.Frame.setTitle  % Frame method
```

The following example uses the form, which fun(a,b,c,...). The response returned from which depends upon the arguments of the function feval. When fun is a function handle, MATLAB evaluates the function using the feval built-in.

```
fun = @abs;
which feval(fun,-2.5)
feval is a built-in function.
```

When fun is the inline function, MATLAB evaluates the function using the feval method of the inline class.

```
fun = inline('abs(x)');
which feval(fun,-2.5)
matlabroot\toolbox\matlab\funfun\@inline\feval.m  % inline
method
```

When you specify an output variable, which returns a cell array of strings to the variable. You must use the *function* form of which, enclosing all arguments in parentheses and single quotes.

```
s = which('private/stradd','-all');
whos s
  Name      Size          Bytes  Class
  s         3x1             562  cell array
Grand total is 146 elements using 562 bytes
```

See Also `dir`, `doc`, `exist`, `lookfor`, `path`, `type`, `what`, `who`

Purpose Repeat statements an indefinite number of times

Syntax

```
while expression
    statements
end
```

Description `while` repeats `statements` an indefinite number of times. The `statements` are executed while the real part of *`expression`* has all nonzero elements. *`expression`* is usually of the form

```
expression rel_op expression
```

where *`rel_op`* is ==, <, >, <=, >=, or ~=.

The scope of a `while` statement is always terminated with a matching `end`.

Arguments

expression

`expression` is a MATLAB expression, usually consisting of variables or smaller expressions joined by relational operators (e.g., `count < limit`), or logical functions (e.g., `isreal(A)`).

Simple expressions can be combined by logical operators (&,|,~) into compound expressions such as the following. MATLAB evaluates compound expressions from left to right, adhering to operator precedence rules.

```
(count < limit) & ((height - offset) >= 0)
```

statements

`statements` is one or more MATLAB statements to be executed only while the *`expression`* is `true` or nonzero.

Remarks

Nonscalar Expressions

If the evaluated `expression` yields a nonscalar value, then every element of this value must be `true` or nonzero for the entire expression to be considered `true`. For example, the statement, `while (A < B)` is `true` only if each element of matrix `A` is less than its corresponding element in matrix `B`. See Example 2, below.

Partial Evaluation of the Expression Argument

Within the context of an `if` or `while` expression, MATLAB does not necessarily evaluate all parts of a logical expression. In some cases it is possible, and often advantageous, to determine whether an expression is `true` or `false` through only partial evaluation.

For example, if `A` equals zero in statement 1 below, then the expression evaluates to `false`, regardless of the value of `B`. In this case, there is no need to evaluate `B` and MATLAB does not do so. In statement 2, if `A` is nonzero, then the expression is `true`, regardless of `B`. Again, MATLAB does not evaluate the latter part of the expression.

```
1)   while (A & B)                 2)   while (A | B)
```

You can use this property to your advantage to cause MATLAB to evaluate a part of an expression only if a preceding part evaluates to the desired state. Here are some examples.

```
while (b ~= 0) & (a/b > 18.5)

if exist('myfun.m') & (myfun(x) >= y)

if iscell(A) & all(cellfun('isreal', A))
```

Examples

Example 1 - Simple `while` Statement

The variable eps is a tolerance used to determine such things as near singularity and rank. Its initial value is the *machine epsilon*, the distance from 1.0 to the next largest floating-point number on your machine. Its calculation demonstrates `while` loops.

```
eps = 1;
while (1+eps) > 1
    eps = eps/2;
end
eps = eps*2
```

Example 2 - Nonscalar Expression

Given matrices A and B

```
A =                  B =
     1     0              1     1
     2     3              3     4
```

Expression	Evaluates As	Because
A < B	false	A(1,1) is not less than B(1,1).
A < (B + 1)	true	Every element of A is less than that same element of B with 1 added.
A & B	false	A(1,2) & B(1,2) is false.
B < 5	true	Every element of B is less than 5.

See Also

if, for, end, all, any, break, return, switch

whitebg

Purpose

Change axes background color

Syntax

```
whitebg
whitebg(h)
whitebg(ColorSpec)
whitebg(h,ColorSpec)
```

Description

`whitebg` complements the colors in the current figure.

`whitebg(h)` complements colors in all figures specified in the vector `h`.

`whitebg(ColorSpec)` and `whitebg(h,ColorSpec)` change the color of the axes, which are children of the figure, to the color specified by `ColorSpec`.

Remarks

`whitebg` changes the colors of the figure's children, with the exception of shaded surfaces. This ensures that all objects are visible against the new background color. `whitebg` sets the default properties on the root such that all subsequent figures use the new background color.

Examples

Set the background color to blue-gray.

```
whitebg([0 .5 .6])
```

Set the background color to blue.

```
whitebg('blue')
```

See Also

`ColorSpec`

The figure graphics object property `InvertHardCopy`.

Purpose

List variables in the workspace

Graphical Interface

As an alternative to `whos`, use the Workspace browser. To open it, select **Workspace** from the **View** menu in the MATLAB desktop.

Syntax

```
who
whos
who('global')
whos('global')
who('-file','filename')
whos('-file','filename')
who('var1','var2',...)
who('-file','filename','var1','var2',...)
s = who(...)
s = whos(...)
who -file filename var1 var2 ...
whos -file filename var1 var2 ...
```

Description

`who` lists the variables currently in the workspace.

`whos` lists the current variables and their sizes and types. It also reports the totals for sizes.

`who('global')` and `whos('global')` list the variables in the global workspace.

`who('-file','filename')` and `whos('-file','filename')` list the variables in the specified MAT-file `filename`. Use the full path for `filename`.

`who('var1','var2',...)` and `whos('var1','var2',...)` restrict the display to the variables specified. The wildcard character `*` can be used to display variables that match a pattern. For example, `who('A*')` finds all variables in the current workspace that start with `A`.

`who('-file','filename','var1','var2',...)` and `whos('-file','filename','var1','var2',...)` list the specified variables in the MAT-file `filename`. The wildcard character `*` can be used to display variables that match a pattern.

`s = who(...)` returns a cell array containing the names of the variables in the workspace or file and assigns it to the variable `s`.

`s = whos(...)` returns a structure with these fields

`name`	`variable name`
`size`	`variable size`
`bytes`	`number of bytes allocated for the array`
`class`	`class of variable`

and assigns it to the variable `s`.

`who` **`-file`** `filename var1 var2 ...` and `whos` **`-file`** `filename var1 var2 ...` are the unquoted forms of the syntax.

See Also

`assignin`, `dir`, `evalin`, `exist`, `what`, `workspace`

Purpose

Wilkinson's eigenvalue test matrix

Syntax

```
W = wilkinson(n)
```

Description

`W = wilkinson(n)` returns one of J. H. Wilkinson's eigenvalue test matrices. It is a symmetric, tridiagonal matrix with pairs of nearly, but not exactly, equal eigenvalues.

Examples

```
wilkinson(7)

ans =

    3    1    0    0    0    0    0
    1    2    1    0    0    0    0
    0    1    1    1    0    0    0
    0    0    1    0    1    0    0
    0    0    0    1    1    1    0
    0    0    0    0    1    2    1
    0    0    0    0    0    1    3
```

The most frequently used case is `wilkinson(21)`. Its two largest eigenvalues are both about 10.746; they agree to 14, but not to 15, decimal places.

See Also

`eig`, `gallery`, `pascal`

wk1read

Purpose

Read Lotus123 spreadsheet file (`.wk1`)

Syntax

```
M = wk1read(filename)
M = wk1read(filename,r,c)
M = wk1read(filename,r,c,range)
```

Description

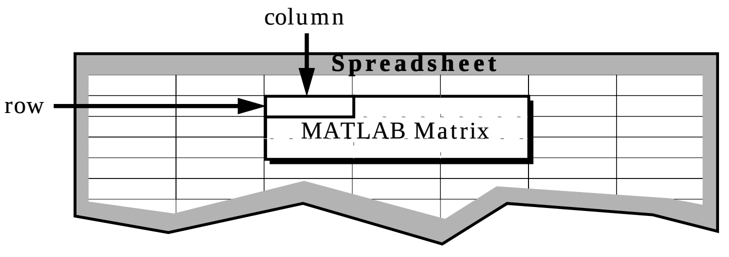
`M = wk1read(filename)` reads a Lotus123 WK1 spreadsheet file into the matrix `M`.

`M = wk1read(filename,r,c)` starts reading at the row-column cell offset specified by `(r,c)`. `r` and `c` are zero based so that `r=0`, `c=0` specifies the first value in the file.

`M = wk1read(filename,r,c,range)` reads the range of values specified by the parameter `range`, where `range` can be:

- A four-element vector specifying the cell range in the format

  `[upper_left_row upper_left_col lower_right_row lower_right_col]`



- A cell range specified as a string; for example, `'A1...C5'`.
- A named range specified as a string; for example, `'Sales'`.

See Also

`wk1write`

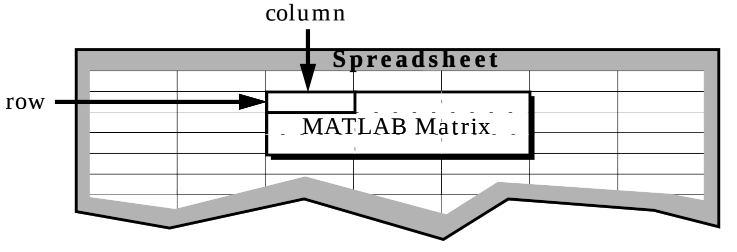
Purpose Write a matrix to a Lotus123 WK1 spreadsheet file

Syntax

```
wk1write(filename,M)
wk1write(filename,M,r,c)
```

Description `wk1write(filename,M)` writes the matrix `M` into a Lotus123 WK1 spreadsheet file named filename.

`wk1write(filename,M,r,c)` writes the matrix starting at the spreadsheet location `(r,c)`. `r` and `c` are zero based so that `r=0`, `c=0` specifies the first cell in the spreadsheet.



See Also `wk1read`

workspace

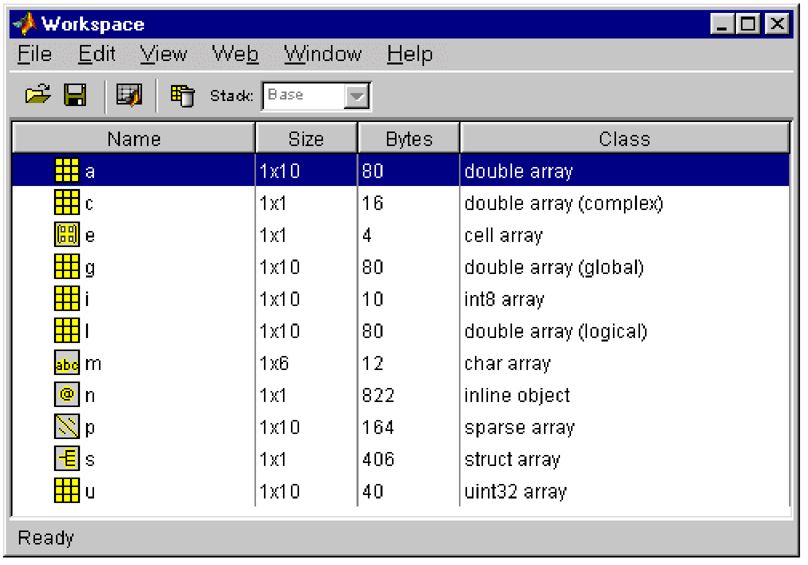
Purpose Display the Workspace browser, a tool for managing the workspace

Graphical Interface As an alternative to the `workspace` function, select **Workspace** from the **View** menu in the MATLAB desktop.

Syntax

```
workspace
```

Description `workspace` displays the Workspace browser, a graphical user interface that allows you to view and manage the contents of the MATLAB workspace. It provides a graphical representation of the `whos` display, and allows you to perform the equivalent of the `clear`, `load`, `open`, and `save` functions.



To see and edit a graphical representation of a variable, double-click the variable in the Workspace browser. The variable is displayed in the Array Editor, where you can edit it. You can only use this feature with numeric arrays.

See Also `who`

Purpose

Label the *x*-, *y*-, and *z*-axis

Syntax

```
xlabel('string')
xlabel(fname)
xlabel(...,'PropertyName',PropertyValue,...)
h = xlabel(...)

ylabel(...)
h = ylabel(...)

zlabel(...)
h = zlabel(...)
```

Description

Each axes graphics object can have one label for the *x*-, *y*-, and *z*-axis. The label appears beneath its respective axis in a two-dimensional plot and to the side or beneath the axis in a three-dimensional plot.

`xlabel('string')` labels the *x*-axis of the current axes.

`xlabel(fname)` evaluates the function `fname`, which must return a string, then displays the string beside the *x*-axis.

`xlabel(...,'PropertName',PropertyValue,...)` specifies property name and property value pairs for the text graphics object created by `xlabel`.

`h = xlabel(...)`, `h = ylabel(...)`, and `h = zlabel(...)` return the handle to the text object used as the label.

`ylabel(...)` and `zlabel(...)` label the *y*-axis and *z*-axis, respectively, of the current axes.

Remarks

Re-issuing an `xlabel`, `ylabel`, or `zlabel` command causes the new label to replace the old label.

For three-dimensional graphics, MATLAB puts the label in the front or side, so that it is never hidden by the plot.

See Also

`text`, `title`

xlim, ylim, zlim

Purpose Set or query axis limits

Syntax Note that the syntax for each of these three functions is the same; only the `xlim` function is used for simplicity. Each operates on the respective x-, y-, or z-axis.

```
xlim
xlim([xmin xmax])
xlim('mode')
xlim('auto')
xlim('manual')
xlim(axes_handle,...)
```

Description `xlim` with no arguments returns the respective limits of the current axes.

`xlim([xmin xmax])` sets the axis limits in the current axes to the specified values.

`xlim('mode')` returns the current value of the axis limits mode, which can be either `auto` (the default) or `manual`.

`xlim('auto')` sets the axis limit mode to `auto`.

`xlim('manual')` sets the respective axis limit mode to `manual`.

`xlim(axes_handle,...)` performs the set or query on the axes identified by the first argument, `axes_handle`. When you do not specify an axes handle, these functions operate on the current axes.

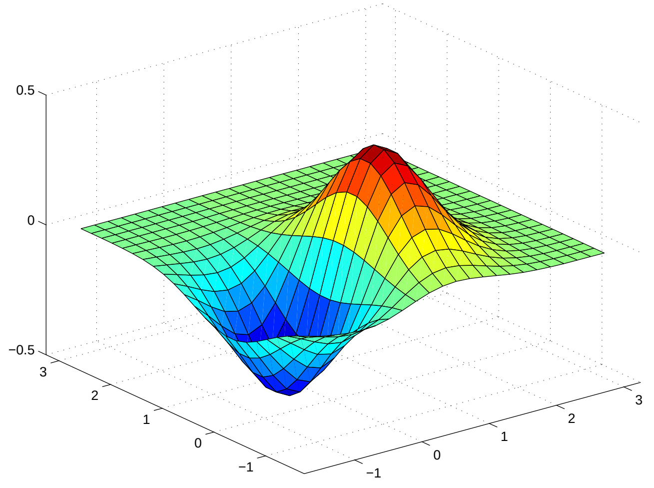
Remarks `xlim`, `ylim`, and `zlim` set or query values of the axes object `XLim`, `YLim`, `ZLim`, and `XLimMode`, `YLimMode`, `ZLimMode` properties.

When the axis limit modes are `auto` (the default), MATLAB uses limits that span the range of the data being displayed and are round numbers. Setting a value for any of the limits also sets the corresponding mode to `manual`. Note that high-level plotting functions like `plot` and `surf` reset both the modes and the limits. If you set the limits on an existing graph and want to maintain these limits while adding more graphs, use the `hold` command.

Examples

This example illustrates how to set the *x*- and *y*-axis limits to match the actual range of the data, rather than the rounded values of `[-2 3]` for the *x*-axis and `[-2 4]` for the *y*-axis originally selected by MATLAB.

```
[x,y] = meshgrid([–1.75:.2:3.25]);
z = x.*exp(–x.^2–y.^2);
surf(x,y,z)
xlim([–1.75 3.25])
ylim([–1.75 3.25])
```



See Also

`axis`

The axes properties `XLim`, `YLim`, `ZLim`

The “Aspect Ratio” section in the online *Using MATLAB Graphics* manual.

xlsfinfo

Purpose Determine if file contains Microsoft Excel (`.xls`) spreadsheet

Syntax

```
[A, Descr] = xlsfinfo('filename')
```

Description `[A, Descr] = xlsfinfo('filename')` returns the character array `'Microsoft Excel Spreadsheet'` in `A` if filename is an Excel spreadsheet. Returns an empty string if `filename` is not an Excel spreadsheet. `Descr` is a cell array of strings containing the name of each spreadsheet in the file.

Examples When `filename` is an Excel spreadsheet:

```
[a,descr] = xlsfinfo('tempdata.xls')

a =

Microsoft Excel Spreadsheet

descr =

    'Sheet1'
```

See Also `xlsread`

Purpose

Read Microsoft Excel spreadsheet file (.xls)

Syntax

```
A = xlsread('filename')
[A, B ] = xlsread('filename')
[...] = xlsread('filename','sheetname')
```

Description

`A = xlsread('filename')` returns numeric data in array `A` from the first sheet in Microsoft Excel spreadsheet file named *filename*. `xlsread` ignores leading rows or columns of text. However, if a cell not in a leading row or column is empty or contains text, `xlsread` puts a `NaN` in its place in `A`.

`[A, B]= xlsread('filename')` returns numeric data in array `A`, text data in cell array `B`. If the spreadsheet contains leading rows or columns of text, `xlsread` returns only those cells in `B`. If the spreadsheet contains text that is not in a row or column header, `xlsread` returns a cell array the same size as the original spreadsheet with text strings in the cells that correspond to text in the original spreadsheet. All cells that correspond to numeric data are empty.

`[...]= xlsread('filename','sheetname')` read sheet specified in `sheetname`. Returns an error if `sheetname` does not exist. To determine the names of the sheets in a spreadsheet file, use `xlsfinfo`.

Handling Excel Date Values

When reading date fields from Excel files, you must convert the Excel date values into MATLAB date values. Both Microsoft Excel and MATLAB represent dates as serial days elapsed from some reference date. However, Microsoft Excel uses January 1, 1900 as the reference date and MATLAB uses January 1, 0000.

For example, if your Excel file contains these date values,

```
4/12/00
4/13/00
4/14/00
```

use this code to convert the dates to MATLAB dates.

```
excelDates = xlsread('filename')
matlabDates = datenum('30-Dec-1899') + excelDates
datestr(matlabDates,2)
ans =
```

```
04/12/00
04/13/00
04/14/00
```

Examples

Example 1 – Reading Numeric Data

The Microsoft Excel spreadsheet file, `testdata1.xls`, contains this data:

```
    1    6
    2    7
    3    8
    4    9
    5   10
```

To read this data into MATLAB, use this command:

```
A = xlsread('testdata1.xls')
A =
    1    6
    2    7
    3    8
    4    9
    5   10
```

Example 2 – Handling Text Data

The Microsoft Excel spreadsheet file, `testdata2.xls`, contains a mix of numeric and text data.

```
    1    6
    2    7
    3    8
    4    9
    5   text
```

`xlsread` puts a `NaN` in place of the text data in the result.

```
A = xlsread('testdata2.xls')
A =
     1     6
     2     7
     3     8
     4     9
     5   NaN
```

Example 3 – Handling Files with Row or Column Headers

The Microsoft Excel spreadsheet file, `tempdata.xls`, contains two columns of numeric data with text headers for each column:

```
Time  Temp
12      98
13      99
14      97
```

If you want to import only the numeric data, use `xlsread` with a single return argument. `xlsread` ignores a leading row or column of text in the numeric result.

```
ndata = xlsread('tempdata.xls')

ndata =

    12    98
    13    99
    14    97
```

To import both the numeric data and the text data, specify two return values for `xlsread`.

```
[ndata, headertext] = xlsread('tempdata.xls')
ndata =

    12    98
    13    99
    14    97

headertext =

    'time'    'temp'
```

See Also

`wk1read`, `textread`, `xlsfinfo`

Purpose Exclusive or

Syntax

```
C = xor(A,B)
```

Description `C = xor(A,B)` performs an exclusive OR operation on the corresponding elements of arrays `A` and `B`. The resulting element `C(i,j,...)` is logical true (`1`) if `A(i,j,...)` or `B(i,j,...)`, but not both, is nonzero.

A	B	C
zero	zero	0
zero	nonzero	1
nonzero	zero	1
nonzero	nonzero	0

Examples Given `A = [0 0 pi eps]` and `B = [0 -2.4 0 1]`, then

```
C = xor(A,B)
C =
   0   1   1   0
```

To see where either `A` or `B` has a nonzero element and the other matrix does not,

```
spy(xor(A,B))
```

See Also `all`, `any`, `find`, `logical operators`

zeros

Purpose

Create an array of all zeros

Syntax

```
B = zeros(n)
B = zeros(m,n)
B = zeros([m n])
B = zeros(d1,d2,d3...)
B = zeros([d1 d2 d3...])
B = zeros(size(A))
```

Description

`B = zeros(n)` returns an `n`-by-`n` matrix of zeros. An error message appears if `n` is not a scalar.

`B = zeros(m,n)` or `B = zeros([m n])` returns an `m`-by-`n` matrix of zeros.

`B = zeros(d1,d2,d3...)` or `B = zeros([d1 d2 d3...])` returns an array of zeros with dimensions `d1`-by-`d2`-by-`d3`-by-`...` .

`B = zeros(size(A))` returns an array the same size as `A` consisting of all zeros.

Remarks

The MATLAB language does not have a dimension statement; MATLAB automatically allocates storage for matrices. Nevertheless, for large matrices, MATLAB programs may execute faster if the `zeros` function is used to set aside storage for a matrix whose elements are to be generated one at a time, or a row or column at a time. For example

```
x = zeros(1,n);
for i = 1:n, x(i) = i; end
```

See Also

`eye`, `ones`, `rand`, `randn`

Purpose

Zoom in and out on a 2-D plot

Syntax

```
zoom on
zoom off
zoom out
zoom reset
zoom
zoom xon
zoom yon
zoom(factor)
zoom(fig, option)
```

Description

`zoom on` turns on interactive zooming. When interactive zooming is enabled in a figure, pressing a mouse button while your cursor is within an axes zooms into the point or out from the point beneath the mouse. Zooming changes the axes limits.

- For a single-button mouse, zoom in by pressing the mouse button and zoom out by simultaneously pressing **Shift** and the mouse button.
- For a two- or three-button mouse, zoom in by pressing the left mouse button and zoom out by pressing the right mouse button.

Clicking and dragging over an axes when interactive zooming is enabled draws a rubber-band box. When the mouse button is released, the axes zoom in to the region enclosed by the rubber-band box.

Double-clicking over an axes returns the axes to its initial zoom setting.

`zoom off` turns interactive zooming off.

`zoom out` returns the plot to its initial zoom setting.

`zoom reset` remembers the current zoom setting as the initial zoom setting. Later calls to `zoom out`, or double-clicks when interactive `zoom` mode is enabled, will return to this zoom level.

`zoom` toggles the interactive zoom status.

`zoom xon` and `zoom yon` set `zoom on` for the *x*- and *y*-axis, respectively.

`zoom(factor)` zooms in or out by the specified zoom factor, without affecting the interactive zoom mode. Values greater than 1 zoom in by that amount, while numbers greater than 0 and less than 1 zoom out by `1/factor`.

`zoom(fig, option)` Any of the above options can be specified on a figure other than the current figure using this syntax.

Remarks

`zoom` changes the axes limits by a factor of two (in or out) each time you press the mouse button while the cursor is within an axes. You can also click and drag the mouse to define a zoom area, or double-click to return to the initial zoom level.

Index

D

G

Q

R

V